An Object-Oriented Repository-based Software Synthesis System

THESIS

Gary L. Cornn, Jr., Captain, USAF

AFIT/GCS/ENG/00M-05

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U. S. Government.

AFIT/GCS/ENG/00M-05

An Object-Oriented Repository-based Software Synthesis System

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science

Gary L. Cornn, Jr., B.S.

Captain, USAF

March, 2000

AFIT/GCS/ENG/00M-05

An Object-Oriented Repository-based Software Synthesis System

Gary L. Cornn, Jr., B.S.

Captain, USAF

Approved:

Robert P. Graham Jr.	9 MAR 2000
Maj. Robert P. Graham, Jr. Committee Chair	Date
Thomas C. Hartrum	9 Mar 2000
Dr. Thomas C. Hartrum Committee Member	Date
Michael L. Talbert	9 Mar 2000
Maj. Michael L. Talbert Committee Member	Date

Acknowledgements

Now to him who is able to do immeasurably more than all we ask or imagine, according to his power that is at work within us, to him be glory in the church and in Christ Jesus throughout all generations, for ever and ever! Amen.

Ephesians 3:20,21

I'm forever grateful to my family, friends, instructors, and "lab-mates" for their help and support during my time at AFIT. To my advisor, Maj. Graham, I say "thanks" for his encouragement and motivation during my research. I also would like to thank my committee members, Maj. Talbert and Dr. Hartrum, for their added wisdom and guidance.

Most of all, I offer my thanks to my wife, Cheri and girls, Tiffany and Allison, for their love, support, understanding, and patience during this 18 month process. I love you—now it's time to get your husband and dad back.

Gary L. Cornn, Jr.

Table of Contents

List of Figures

List of Tables

Abstract

This research provides a repository on which various Air Force Institute of Technology (AFIT) transformational software synthesis tools can store, share, and manage data using a common repository information model. This information model was created by integrating a variety of separately-developed AFIT software synthesis object models into a "wide-spectrum" model. Additionally, a methodology for describing complex relationships between artifacts in the repository is described. These relationships can be used to relate software synthesis artifacts created in a variety of formats, including text, binary, and the AFIT Wide-Spectrum Object Modeling Environment (AWSOME) information model. The relationships can be exploited for the retrieval, understanding, and selection of reusable software engineering artifacts. Finally, a methodology that uses the repository relationships to generate a history of the semi-automatically generated designs is described. Future efforts can use the design history to re-create designs automatically when new requirements dictate changes to a related analysis model.

An Object-Oriented Repository-based Software Synthesis System

I. Introduction

The software engineering community has long touted reusability as a fundamental aspect of improving software and shortening development cycles. One enabling technology is reuse libraries or catalogs for reusable source code. More recently, researchers and tool developers have recognized that the use of multiple CASE tools, pre-existing COTS libraries, and other products of the software development process requires new ways of thinking about cataloging these artifacts. Users need to quickly screen, identify, and decide on the potential reusability of the artifacts. In addition, software developers need a way to organize, manage, retrieve, and share persistent data of software tools. This need has driven research and development of repository systems.

Knowledge-based Software Engineering(KBSE) is an area of software engineering where reuse is essential. KBSE is not concerned with code reuse, but with the reuse of knowledge. In AFIT KBSE research, domain knowledge is developed and specified in a formal language, then reused to generate domain-specific software automatically. In a large software development environment, domain models and many other products of the process need to be organized, managed, retrieved, and shared by KBSE tools.

1.1 Problem

Over the past few years, AFIT researchers have developed and implemented KBSE tools. The tools synthesize software by transforming formal domain knowledge into formal

specifications, specifications into designs, and designs into source code [26]. Currently, the tool accepts Z specifications as input and successfully transforms primitive [32] and aggregate [47] classes to Ada or, recently, C++. The tool's data structure is an object model in the form of an abstract syntax tree (AST). Successive manipulations of the AST transform the specification AST into a design AST. Once the design AST is in place, source code can be generated by further transformation into a language-specific AST or by appropriate "tree-walking" functions.

The current set of tools, collectively named AFITTool, implements the process shown in Figure 1. As can be seen in the figure, the process is designed to use a number of libraries. These libraries are the foundation for reusability in the KBSE environment. Technology for effective and efficient implementation of these libraries can be explored.

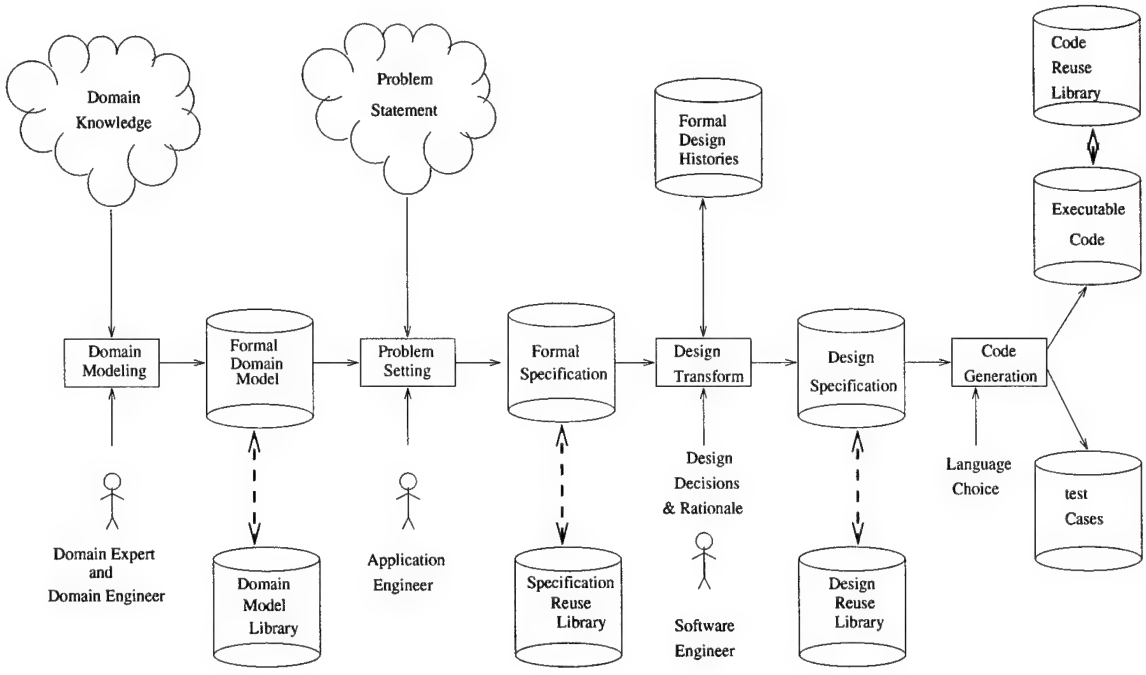

Figure 1. AFITTool Process Flow

For such a system, a large amount of data is created and reused. Primarily, the system must allow the creation of— and have ready access to—domain knowledge. A variety of formally specified domain knowledge is necessary to develop robust formal specifications that can be transformed into source code. In addition, class libraries from many problem domains, such as user interfaces, already exist. These class libraries must be integrated with other domain knowledge to generate an application. Also, existing design expertise is available via design patterns, software architectures, or by creating design information in a CASE tool. This design information is useful as a model for transforming specifications into a design. Additionally, software engineers want to retain and reuse work on current and previous projects. Finally, source code may be generated in a variety of target languages or for a number of heterogeneous platforms. Research is needed to determine which KBSE data is reusable and how to enhance its reusability using meta-data such as descriptions and relationships between artifacts.

The many products, or artifacts, of the software synthesis process suffer from the same problem as any software development—change. Domain models, requirements specifications, and system architecture are subject to maintenance due to changing requirements, or system upgrades. For example, new or more detailed information can enhance existing domain knowledge, a user may require new functionality, or systems can migrate from centralized to distributed architectures. This potential for change can require many alternate specifications or designs.

These alternative designs have additional impact in KBSE. In a system where designs are built automatically or semi-automatically, these designs (and design decisions) must be "self documenting." When maintenance is necessary, as mentioned above, the revised

requirements specifications can be re-implemented by "replaying" the saved sequence of transformations to produce a new implementation [3].

Problem Statement: No efficient way to store, manage, relate, and reuse various artifacts of the KBSE process exists. Research is needed to determine the states of the model that are to be stored as artifacts and how to represent the relationships between these artifacts.

1.2 Past Effort

Reusability, as applied to software, has two chief objectives: 1) don't recreate that which already exists, and 2) build systems using well-defined and verified building blocks [30]. That is also the goal of KBSE. Existing domain knowledge is synthesized into software by a computer. Computer-aided software development is not new. Many software engineering tools support automatic generation of software systems. For example, several tools generate application systems automatically by allowing a user to informally specify the system using some tool-specific syntax or domain-specific language.

In some domains, tools of this nature have had commercial success. For example, several tools on the market allow a user to specify relational databases and associated graphical interfaces for data entry and retrieval. These tools then automatically generate the data definition language (DDL), associated graphical interfaces, and data manipulation language (DML) for database population, browsing, retrieval, modification, and report generation. However, there are no general purpose tools capable of generating application systems automatically for a wide range of domains. Tools of this kind are still the subject of investigation.

The need to catalog and manage the many products of the software development process led to research in repository systems. A number of research efforts have produced reuse libraries and repository systems [5, 9, 13]. They contain research on representation and associated searching methods, repository architectures, and repository uses.

Repositories have been proposed for use in areas such as software engineering, data warehousing, and managing web sites. Repositories are used for managing meta-data about objects in the repository. For example, relationships between objects are meta-data. This meta-data can be searched, allowing users to easily locate and reuse artifacts. The "searchability" of the meta-data depends upon the representation method used.

A representation method describes the artifacts and the relationships between them. The relationships range from library classification, to hypertext-based, to those that are predicated on formal languages. Founded in the field of information retrieval, representation methods and searching techniques are an essential provision of a repository.

Repository researchers have described a general architecture for repositories [7]. Artifacts in the repository are modeled in the top layer of a repository architecture. This layer is called the information model. From the perspective of tool developers, the information model is the tool's object model. To database developers, the information model is the database schema. The other two layers of the architecture are the repository engine and the underlying database system. The underlying database is the physical store of the repository. Between the database system and the information model is the repository engine. The repository engine provides necessary functionality such as management and extensibility of repository objects.

Since software tools (like most software systems developed in the current era) are generally object-oriented, the use of a persistent object store should provide a natural implementation platform for a repository. However, according to one expert [7], an object-oriented database alone cannot serve as a repository. Some proprietary repositories and research systems attempt to use object-oriented database management systems, but do not implement all the desired repository engine features. Even one of the most well known commercial repositories, the Microsoft Repository [5], uses a relational database as its underlying database management system.

1.3 Research

Since one use of a repository is to manage data for software development tools, a properly designed repository system can fulfill the reuse goals of AFIT's KBSE environment. This research pursued the use of a repository to achieve the reuse goals. Specifically, this effort devised a wide-spectrum object model for software synthesis and adapted this model to a repository information model. The research effort also provided basic functionality for representing artifacts within the repository by describing the relationships between artifacts. Furthermore, the research demonstrated how these relationships could be used to capture the history of a particular design. These histories can be used to "replay" a design automatically when maintenance is performed on a requirement specification.

1.3.1 Scope. A methodology for combining related AFITTool models was developed. The method was applied to separate domain and design models to create a single wide-spectrum model. The resulting model was redefined as an appropriate infor-

mation model for the AFIT software synthesis repository. This information model was implemented on top of an object-oriented database management system (OODBMS) that served as the repository database. The OODBMS provides a number of functions typical of a repository engine. The research showed how the representation of relationships within the repository could be used to promote reusability in this environment. Finally, the repository relationships were used to develop a methodology for retaining a history of a design. Other repository engine features, such as representing workflow models, versions, and notification were left for future extension of the repository implementation.

1.4 Contributions

This research suggests a new repository-based software synthesis process. It identifies AST states that should be retained in the repository as artifacts, and those states that are transitional forms of the AST. It develops artifact meta-data and a methodology for representing the relationships between repository artifacts. Finally, the research shows how the identified methodology for repository relationships can be used to solve an AFIT KBSE shortcoming: creating design histories.

1.5 Outline

The remaining five chapters provide necessary background, methodology, and implementation details of a model for software synthesis, as well as a basic repository, repository relationships, and results of this research. Chapter II provides necessary background on repository technology, methodologies used for representing relationships (used in the field of information retrieval), and background on KBSE research at AFIT. Chapter III reviews

specific requirements this research attempted to meet, as well as the methodology proposed to meet those requirements. Chapter IV describes how a common model for software synthesis was developed, and Chapter V discusses the particulars of the prototype repository implementation—including some difficulties encountered along the way. Finally, Chapter VI reviews the research goals accomplished by this effort and outlines possible future work.

II. Background

The definition of a repository is almost intuitive—a database of information about engineered artifacts. A repository is further defined as a database shared by software tools. It provides the ability for multiple software tools to share information within their user domain. For example, Boeing has a repository of aircraft design information. This design information consists of persistent data from Computer Aided Design software, as well as additional data about the designs [49].

Researchers have a variety of views of the composition, architecture, and use of a repository and have applied significant effort in these areas. They have also explored the relative importance of this technology in a variety of uses.

This chapter reviews the current state of repository technology and provides necessary background for this research effort. The first section provides an overview of the kinds of artifacts in a repository and their application in a variety of domains. This is followed by a discussion of the organization of a repository and the search capability these representations provide. The third section contains a high-level overview of a repository architecture. Next, the chapter discusses software synthesis research and the various object models and tools currently used in AFIT's environment. Finally, there is a discussion of the automatic generation of software design information.

2.1 Repository Artifacts

Meta-data is "data about data." Repositories are used to manage meta-data for software tools. In practice, anything lending itself to description by meta-data can be

modeled in a repository. Already, researchers have explored the application of repository technology in a number of domains. For example,

- Business process re-engineering uses large process models and information models—both of which need to be managed [6].
- Data warehouses store data from various, interrelated sources. Meta-data about information, such as transformations between the data warehouse schema and external data source formats [8], are being managed by a repository.
- Manufacturing companies use Computer Aided Design (CAD)/Computer Aided Manufacturing (CAM) systems to manage a variety of technical drawings, manufacturing processes (perhaps using a variety of tools), and associated documentation. The relationships between this data, and the translation between tool-specific data formats can be modeled with meta-data [9].
- Web sites are collections (or perhaps multiple collections) of interrelated HTML documents, Java applets, and other documents. Repositories are being used to manage this data [6].
- The DoD has many simulation systems and thousands of simulation scenarios with a range of associated data, from geographical information to performance characteristics of military hardware. This data and its relationship to other simulation data are best modeled as meta-data [12, 37].
- The object-oriented software development process produces models of software containing hundreds of objects with many complex relationships. Data about these

objects and the relationships between them are meta-data. Repositories are well suited for managing this meta-data [9].

- Computer Aided Software Engineering (CASE) tools have varied functions and many different tools may be used on the same project. In today's software development environment, it is likely that a number of tools will participate in the development of a single component. Providing a common database to store and model relationships between the persistent data of the different tools is a job for a repository [13].

Repository artifacts originate from a number of sources. They are acquired through original design, from legacy information [18], or from third party sources. The development of original artifacts will require many aspects of data management such as version and configuration management. Legacy artifacts can be re-engineered or characterized with meta-data for insertion into a newly acquired repository. In addition, components acquired from a commercial source such as *Sun* or *Microsoft* can be added to a repository. For example, a software developer may have access to a number of graphical interface components such as Java's Abstract Windowing Toolkit (AWT) or Microsoft Foundation Classes.

Once the reusable components are acquired, the method by which they make their way into the repository generally follows a standard process: Certify, Index, and Store (Figure 2). Certification is the process by which potentially reusable artifacts are identified and approved. Indexing involves the creation of additional meta-data, such as indexes, relationships, etc. Finally, the artifact is stored in the database. Ultimately, these components will be searched and retrieved, then integrated into new products or processes.

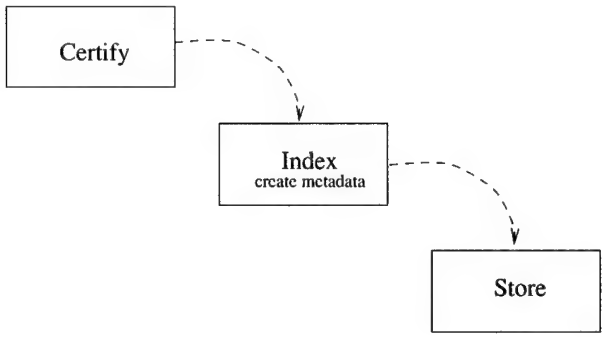

Figure 2. Repository Population

2.1.1 Software Artifacts. Of particular interest is the application of repository technology to the field of software engineering. Many software engineering artifacts are appropriate for management within a repository—not just source code or executables.

Software engineering artifacts can be any product of the software development process. This includes software requirements [4], software design data [43], source or executable code, and associated documentation. Requirements can be developed informally, specified in a formal language such as Z, or can be the persistent data of a CASE tool. Software design data can include design patterns, software or system architectures, and other design documentation. Next, there is program source or executable code. It can exist in multiple programming languages. A system may also have several, functionally equivalent executables for a variety of platforms. In addition, other artifacts such as user manuals or test data will be associated with artifacts in the repository. Finally, new requirements or upgraded system components such as the operating system or database will lead to new versions of many or all these artifacts.

Naturally, this data is modeled in the various CASE tools used in the process. The models are meta-data about these artifacts. This meta-data can be stored in a repository

and shared by multiple tools. Since human users must be able to locate, understand, and reuse the repository artifacts, additional meta-data will be needed to model the many complex relationships between and describe their contents.

2.1.2 Artifact Meta-data. A repository can use meta-data to perform a number of required repository functions. First, meta-data is useful for tracking changes in repository artifacts: software maintenance tasks require new versions to be created or developers may use an existing element of the repository as the basis for creating an entirely new artifact. A system may also require multiple versions of the same code. For example, a Windows 95 and X-windows version of a user interface may exist for a software system. Second, meta-data can be helpful in signaling related artifacts that a change was made. It can trigger automated or software engineer assisted modification of related objects [12]. A simple illustration is the unix makefile. A makefile allows the *make* command to determine when associated source code must be recompiled, and when recompilation is unnecessary.

A number of past efforts have attempted to find the best way to represent meta-data relationships and apply associated searching techniques. The techniques range from those based in the field of information retrieval to formal methods based in lattice theory. These representation and searching methods are the subject of the next section.

2.2 Representation and Searching Methodologies

Over the years, researchers have proposed various ways to classify and represent artifacts within a repository. These methods are intended to promote understanding during searching.

Searching and browsing are highly dependent upon the representation method used. A repository organized as hypertext links will not effectively be searched using a search method based strictly upon keywords. If only keywords are returned from the search, the hypertext link associated with related artifacts will be lost. That notwithstanding, there still appears to be some overlap between search methodologies. This overlap is an attempt by researchers to achieve maximum performance–retrieving the "best" items with the most expediency. In fact, from a repository user's perspective, the "most salient feature" is the effectiveness of the retrieval algorithm [34].

Information retrieval authors define algorithm effectiveness using three criteria: precision, recall, and response time. Precision is defined as the ratio of relevant components retrieved to the overall number of retrieved components. Recall is the number of relevant components retrieved compared to the total number of components in the library. Finally, response time is further subdivided into the number of inspected library components in any given search and the average time a search takes. Time complexity is somewhat intuitive. Users want a search to return results quickly.

As the "most salient feature," one of the most researched areas in repository technology appears to be searching algorithms. Since searches are based on retrieval methods and retrieval methods depend upon the representation method, we can divide searching techniques into the same broad categories as we do representation methods: library sciences, artificial intelligence, hypertext, and formal methods [19,34]. Each one of these categories is presented below.

Table 1. An Example Faceted Classification of typical Unix commands [20]

Tool	Object	Operation	Activity
mkdir	directory	create	
ln	directory	create	
rmdir	directory	destroy	
ed	file	create/modify	edit
vi	file	create/modify	edit

2.2.1 Library Classification. The library classification method is rooted in the field of library sciences. Faceted classification is a typical method libraries used to organize a collection. In faceted classification common terms are identified, grouped by common characteristics (a process subjective to the one conducting the classification), then names are given to the groupings. An example of a faceted classification scheme for Unix commands is shown in Table 1.

2.2.1.1 Representation. Prieto-Diaz [40] devised a way to apply library science methods to domain analysis. His process encompasses three main activities: 1) Identification of Objects and Operations, 2) Abstraction, and 3) Classification.

Once appropriate objects and operations have been selected and identified, common characteristics of groups or classes of groups are selected. These characteristics, or attributes, (the facets in library science terminology) are then placed in frames. The frames represent the objects or groups of objects, and the relationships between them. In the final step, the relationships between the frames can be used to define a classification scheme. The bulk of [40] identifies how this classification scheme can be used for the classification of reusable objects, giving a detailed breakdown of the process, including the input and output of each step.

This is only one possible method based in library sciences. Others are described in [20] and are summarized in Table 2.

Table 2. Other library science–based methods [20]

Enumerated	Highly structured, easy to understand requires extensive domain analysis and limits kinds of relationships
Faceted	Classification and ordering of facets allows complex relationships, but still requires extensive domain analysis
Attribute-valued	Described by a set of attributes and values and values. Similar to faceted but there is no ordering of attribute values
Free text keyword	Terms are automatically extracted from artifacts (e.g. code comments or class interfaces)

Frakes devised a taxonomy of classification methods. It is shown in Figure 3. As can be seen, the methods discussed here all fall under the "controlled" category in the taxonomy.

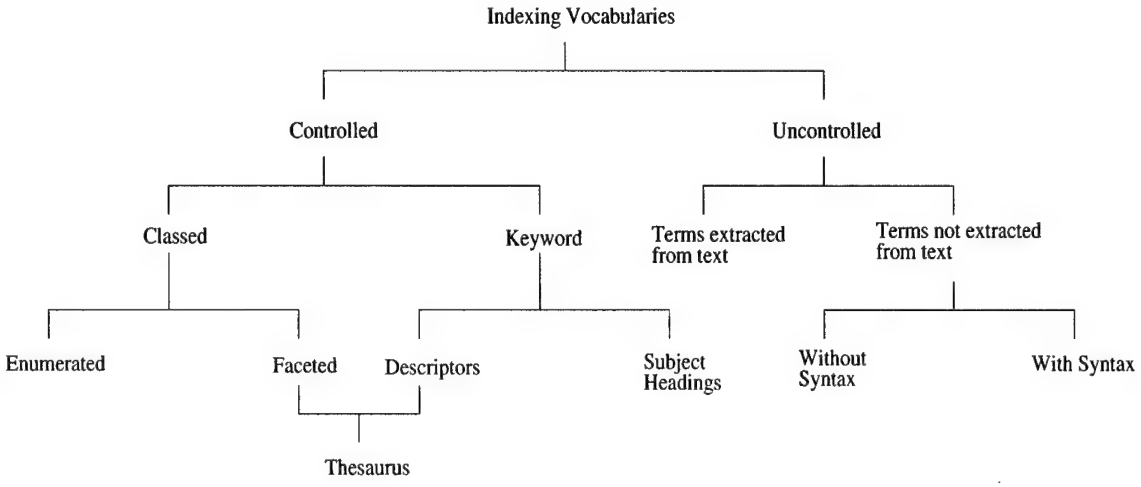

Figure 3. Taxonomy of library science indexing methods [19, 20]

2.2.1.2 Searching. The field of information retrieval is flooded with those investigating searching based upon library science methods. Some past studies conclude that this is the most effective retrieval technique [20]. Most searching techniques in this

area are based on input keywords and retrieval based on those words. The most prolific example of this kind of searching technique is an electronic card catalog in a library. Electronic card catalogs allow users to input a search word such as author, a word from a title, or a pre-defined subject area and searches for results based on exact matches of those key words. Once the search terms are entered, the search method works very much as a query against a database. There are more sophisticated versions, as well. Consider the help facility on any *Microsoft* product. Terms are indexed and as the user types in the search word, the terms appearing in the search results are further refined according to the search criteria entered. This is done interactively but still works very much as a simple exact word match.

These very basic searches have given way to more sophisticated representations for information retrieval. One of the most prevalent is based on searching concepts found in the field of artificial intelligence.

2.2.2 Knowledge Representation. A number of artificial intelligence techniques have been proposed for the representation of artifacts within a repository. Researchers attempt to base these methods on studies of human cognition—how software engineers solve problems and explore reuse during a project. For example, Sen [43] showed the reuse process as opportunistic with no standard sequence of tasks. Sen suggested any tool supporting the reuse process should allow for the exploitation of this fact.

2.2.2.1 Representation. Knowledge-based techniques are characterized by two main features: representational adequacy and heuristic power [19]. Representational

adequacy is the ability of a model to store information about the object being modeled. Heuristic power represents the capability for inference over a represented model [19].

AI approaches have common themes supporting complex and inexact relationships between artifacts. Generally, everything in the repository is classified based on its *similarity* to something else (often referred to as a *fuzzy* relationship). These similarities are usually represented in part by some form of semantic network (see Figure 4) [13,38].

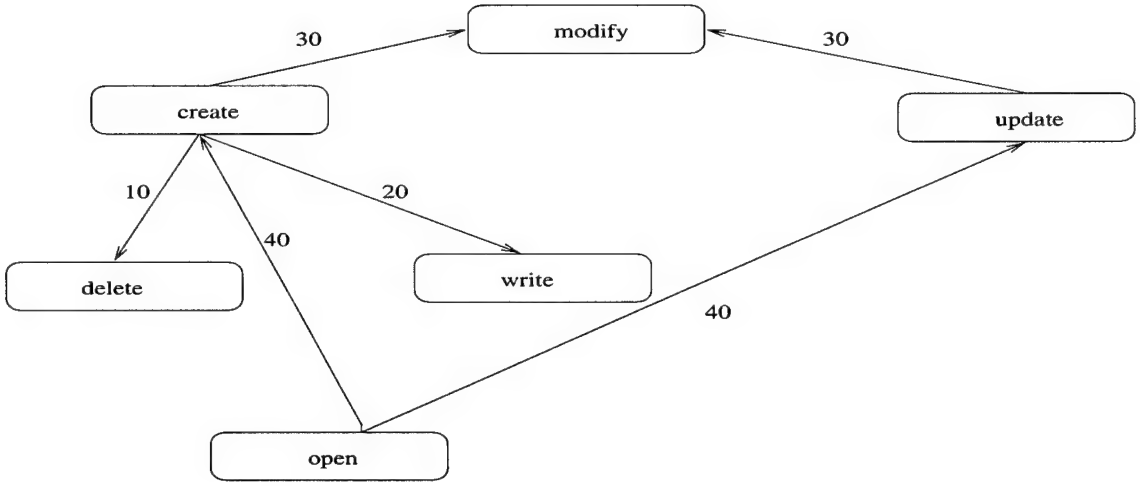

Figure 4. Example of Semantic Net

Semantic networks can be used to model an artifact's identified features as nodes and relationships between artifacts as links. The links can be assigned weights, which represent the similarity between the features. This is often implemented as a tuple called a "feature pair" [38]. AI search techniques will exploit these inexact relationships between features.

2.2.2.2 Searching. Artificial intelligence techniques provide more sophisticated searching algorithms intended to improve recall. AI based methods enhance traditional information retrieval techniques. As in library science-based techniques, some kind of classification and indexing must be in place. Searches return results based on some

similarity of components to the search criteria specified. This similarity is defined in the relationships established within the repository.

As stated previously, these relationships are usually based on some kind of weighting algorithm. The AI based retrieval process used by Ostertag and Diaz [38] selects the best reuse candidate components from a repository based on the degree of similarity between the requested component and the component description existing within the repository. In fact, their repository was organized so this method would search for a package of components. Packages are groupings of related components that have common characteristics. These searches returned packages, or components of packages, which satisfied some or all of the properties in the specified query. In this system, query results could be returned so users could investigate the returned components, or so the results could be used to further improve the search (by allowing a refined search for similar packages).

This process is implemented by storing related components and packages and the weightings within a knowledge base. This approach does not simply use similarity, but allows the search to reason heuristically over the knowledge base about similarity of a given search. These heuristics allow estimation of distances between the indications of similarity based upon weights. The search uses two distance calculations, closeness and subsumption. Subsumption is similar to the concept of inheritance in object-oriented methodologies. When a new component can be created directly from a component or set of existing components it is said to be subsumed by components existing in the repository.

Closeness attempts to compute the likelihood that a new component can be created by modifying, to some degree, another component. If the system *believes* a component

to be modified, the component can be modified again, to create another. When this is true, the component is considered a candidate for reuse. The authors implement this as what they call a feature graph. A feature graph is a weighted, directed graph connecting terms. This feature graph allows the algorithm to estimate closeness in the same way AI techniques attempt to solve any shortest path type of problem. That is, closeness is computed by analyzing the overall distance between any two terms.

This should make the clear the effort that must go into representation methodologies. This kind of searching algorithm would be ineffective if significant effort is not given to the representation of the network of related terms. This amount of "up-front" effort has led researchers to investigate methodologies that might require less work in the beginning while providing adequate results to the user. Some have investigated hypertext-based searching methods as one of these techniques.

2.2.3 Hypertext Methods. One touted benefit of hypertext is the ability to search [13, 29]. Hypertext node types can be based on various criteria such as text output of a design tool or even executable code. Since relationships between these kinds of elements are clearly defined, hypertext links seem well suited for this application.

2.2.3.1 Representation. Proponents of hypertext-based methods offer a number of benefits to this type of organization. Hypertext methods represent reusable artifacts as nodes and relationships as links. For example, code, documentation, and design information can all be related through links to one another.

Proponents also claim hypertext browsing has an added benefit—as users browse the repository, they become more familiar with the components contained therein. As they become more familiar with the repository, they can improve search capability by suggesting improvements to the repository organization. They may also be able to describe how to improve relationships or artifact descriptions in the repository [29]. Hypertext searches, though probably familiar to the reader, are discussed next.

2.2.3.2 Searching. Widespread use of hypertext makes hypertext searching an attractive option for simplifying repository searching. Hyperlinks allow discrete relationships between artifacts in the repository. We see examples of hypertext to organize document repositories today. There are many hypertext search tools available to repository developers. One hypertext-based system attempts to capture the process followed by repository users as they search the repository [31]. The authors believe reuse occurs in three distinct stages: screening, identification, and decision. They developed their system using a hypertext model to support that reuse process.

- Screening - Evaluating a large set of reusable objects to determine a subset for further investigation.
- Identification - Closely examine the subset acquired in screening to determine if any provide the desired functionality.
- Decision - The developer must decide to implement the reusable repository object or build a new one from scratch.

Since almost any user who has searched the internet is aware of the problem of information overload, the authors deal with that problem by suggesting a scripting process.

This scripting process allows the novice users to follow simple scripts, which help search the repository. As users become more experienced with the repository, they may be able to provide more exacting search criteria. Also, some knowledge-based techniques can be present in hypertext searching tools. This can be seen in internet searches which use a thesaurus of similar terms and assign term weights to improve the ability to meet the user's search criteria.

One large European repository research effort, the Software Information Base (SIB) [13] uses a hypertext-based engine for searching and browsing. Constantopoulos, et. al. have attempted to eliminate some of the problems with hypertext with an enhanced user interface that includes a graphical editor and "conceptual modeling facilities." These facilities allow object-oriented relationships (e.g. classification, generalization, aggregation) to be retained and displayed to the system user.

As the size of repositories increase and the relationships of the components within them become more complex, it becomes increasingly difficult to find a reusable component efficiently and accurately. This problem of scalability has been addressed by a number of authors [10, 34]. Some authors expect the more informal methods discussed so far to become less trustworthy as the library size increases. This has led some authors to suggest that software libraries should be organized and searched by means of formal specifications.

2.2.4 Formal Methods. Formal methods are sometimes used to describe software requirements. Additionally, researchers have described existing artifacts using formal algebras as a means of classification [42]. The advantage of formal methods the formal semantics they offer during problem analysis and the provable correctness they provide [25].

2.2.4.1 Representation. The classification of artifacts developed using these formal techniques has been accomplished using discrete mathematics to place artifacts into a specialized partial order known as a lattice. The arrangement of artifacts in the lattice is then part of the meta-data that facilitates the selection of the artifact appropriate for reuse [17, 34].

2.2.4.2 Searching. According to Rym [34], formal searching techniques have some basic premises.

- Software components are represented in the repository by formal specifications.
- Indexing is available to provide some kind of order to allow efficient searching techniques.
- The key does not have to be identical to the search criteria but a match is found when the key *refines* the search criteria (this increases the chance of finding a match).
- If no component refines the search argument, parts of the component can be analyzed.

Artifacts in a formal language-based repository are represented according to an ordering by a lattice structure. A search can consider returning results that are based on the strengths of the component and the lattice. Queries are then specified formally, based on the language in which the components in the library are specified. The retrieval algorithm can then compare the query arguments against nodes in the lattice to determine if any of them match the specification. This is another way to allow results based on inexact (fuzzy) matches.

If the specification refines the query, it is considered a possible match. If the query is also found to refine a descendant of an element in the result set, then the element is deleted from the result set and replaced by the descendant.

The actual implementation of this searching technique is achieved with the use of a theorem prover. First exact matches are computed by proving equivalence. Then approximate matches are found using the lattice property *meet*. A meet is interpreted as the degree of commonality between two specifications.

Rym, *et al.* describe several other proposed formal-based methods. One represents components by predicates describing the main features and interface characteristics. Others are based on signature mapping using polymorphic types. Yet another represents reusable components by describing their signature and by defining axioms that describe interactions between their methods. Finally, formal-based searching methods that restrict the searches to specific domains due to "unpredictability in theorem proving" have been proposed. This technique provides a significant reduction in the search space, but means the library user must be familiar with the search domain.

The area in which formal based methods tend to fall down is not precision. Recall is adequate, as well. Unfortunately, due to the speed of theorem provers, the response time is not likely to encourage people to use this methodology. Theorem provers suffer from the problem of *combinatorial explosion*. Combinatorial explosion occurs because the theorem prover automatically generates any rules it can—many of which may not have utility in deriving a solution. This process continues at each successive level where all possible combinations of rules generated at the previous level are explored. Some of these

combinations will be a step toward proving the solution; however, many will be worthless. As can be seen, this combinatorial nature can make theorem proving—therefore, formal search methods—computationally expensive.

2.2.5 Conclusion. In conclusion, most researchers agree that no one searching method appears to be the panacea for locating reusable artifacts in a repository. In practice, some combinations of searching solutions should be available to allow the most success in selecting reusable artifacts from a repository. This will provide the user with options for searches to use to achieve the best results under a given circumstance [20].

Though the representation methods discussed have been the topic of much research, the library classification methods have gained widest acceptance in practice. Even hypertext and artificial intelligence based approaches usually require some aspect of the library science based schemes to provide labels for nodes and links as appropriate for the methodology. Also, these other methods need some kind of analysis of similarity to *seed* weightings of links (even if the technique will continuously update those weights based on new information).

All of these methods require some form of domain analysis, and some kind of assigned groupings (e.g. facets). Obviously, these representation methods must be implemented in some physical structure. This brings to light the need for architecture for representing systems. This is discussed in the next section.

2.3 Architecture

A number of experimental and commercial repository systems are available. Bernstein [5] describes three main components of a repository architecture; and in addition some associated standard tools (shown in Figure 5). The information model captures common or tool-specific meta-data representing the artifacts in the repository. The repository engine provides the repository with object-oriented capability and other useful functionality. The third component is the actual database system within which the information is stored. Generic repository tools are used to search and administer the repository.

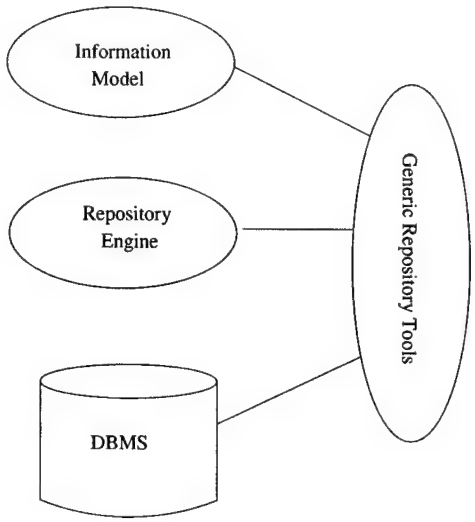

Figure 5. A basic repository architecture

2.3.1 Repository Engine. The repository engine is a layer built on top of a database system. It provides the true functionality of a repository. The features of the repository engine include:

- Object Management - Provides access to the stored state of the object

- Dynamic Extensibility - Ability to extend databases with new types or classes within the information model.
- Relationship Management - Provision of relationship semantics which describe complex relationships between artifacts.
- Notification - Capability to notify users or trigger other operations based on changes to a particular object in the repository.
- Version Management - Management throughout the engineering process of different iterations of a design.
- Configuration Management - Groupings of versioned components into work units for a particular system. Configurations are themselves versioned.

2.3.1.1 Relationships and Descriptions in the Repository Engine. The two chief discussions in the literature of the make-up of a repository's object model take slightly different approaches [5,13]. In a well-known European effort, the authors describe the Software Information Base (SIB). SIB uses a "global SIB structure" that is built in a two-layered structure. The bottom layer contains simple classes stored in a very generic information model. The information model provides common object-oriented concepts such as attribution, aggregation, classes, association, generalization, and specialization. The top layer of the global structure gives users the capability to associate related artifacts from various software development phases using "correspondence relationships."

These relationships are implemented with the use of what they call "application frames." An application frame contains one mandatory implementation—itself associated with a particular language. Additionally, the application frame will contain optional col-

lections of requirements and design information. An example application frame for a hotel information system is shown in Figure 6.

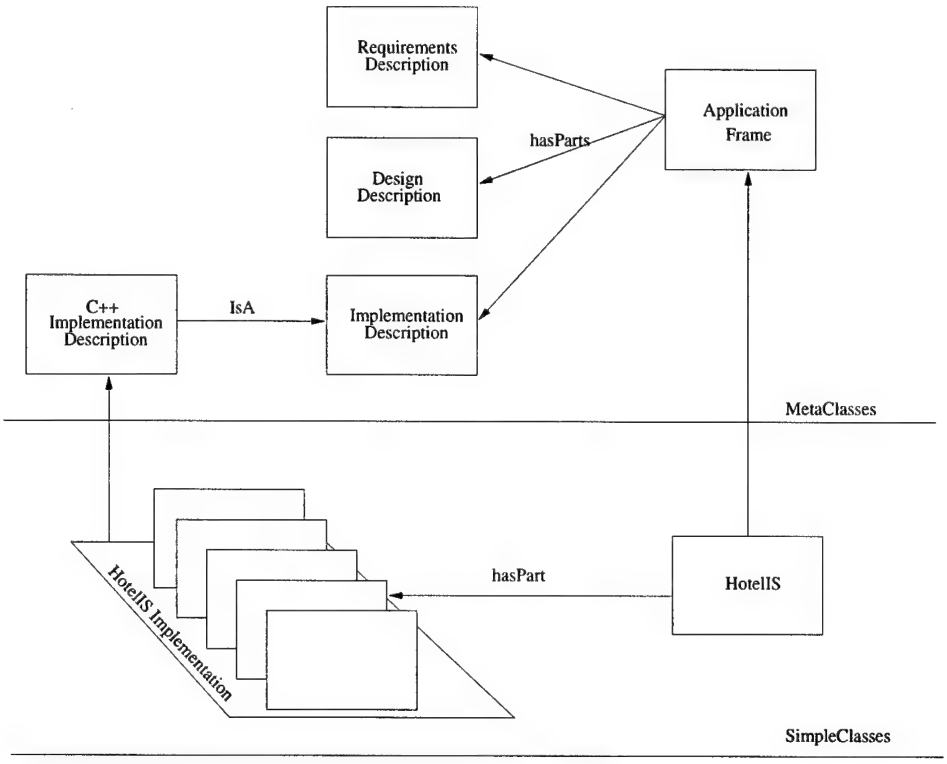

Figure 6. A sample application frame [13]

Bernstein [5] uses Microsoft's Common Object Model (COM) to describe relationships in the Microsoft Repository. Unlike the more software-specific approach of SIB, he describes and relates artifacts in terms of four generic objects implemented as part of the Repository Engine: Repository Session, Repository Object, Relationship Object, and Collection Object. These are explained in Table 3.

Table 3. Repository Object Model [5]

Repository Session	The repository database
Repository Object	A persistent object in the repository
Relationship Object	Connects two relationship objects and has properties of its own—also stored in repository
Collection Object	A set of relationship objects

In the Microsoft Repository, these four objects are used to define and store various object-oriented types in the repository. They can be used to define object classes, relationships (associations), interfaces (includes inheritance), properties (data type with repository-specific meta-data), collections (includes aggregation and roles), and methods. Similar to SIB, this information is used to describe relationships between instances of artifacts represented in a particular information model.

2.3.2 Database Systems. Many of the features of the repository engine (Section 2.3.1) are dependent upon the database system in which the repository engine is implemented. Some of these features can take advantage of the functionality built into an object-oriented database management system. Other capabilities must be explicitly built into the repository engine regardless of the DBMS used.

For example, object management is already a capability built into an object-oriented database management system. Using an OODBMS will allow the trivial implementation of this feature.

Dynamic extensibility allows new types to be added and existing ones to be extended. OO database management systems do not commonly provide all the facilities necessary for this operation. Though OODBMS's allow the schema to be modified with new and extended types, this is only half the job. For example, if a maintenance task requires the modification of a class definition, the repository should provide the capability to convert existing objects' attribute values and aggregate references to the new structure. Implementation of this capability is often left to the application developer. This conversion process

might be best implemented in the repository engine (rather than be implemented by each tool developer).

2.3.3 Information Model. The information model is the schema in which the actual artifacts are modeled. It models the relationships between objects and any other meta-data about the objects. As discussed previously, one application of a repository is to allow use of the same data by multiple tools. For example, Colonese developed a common object-oriented meta-model (and methodology for creating and extending that model) that integrates multiple simulation systems [12]. The Colonese model is a perfect example of an information model applicable to the domain of simulation tools.

In the Microsoft Repository, there is a common information model[1]. It provides a built-in implementation of commonly-used models including Unified Modeling Language (UML), Extensible Markup Language (XML), Object Constraint Language (OCL), and Structured Query Language (SQL).

As can be imagined, information models for tools in the commercial world are proprietary and are often filled with vendor trade secrets. Since these trade secrets provide the competitive advantage of the tool [7], a common information model for a repository simply will not provide a common representation across multiple vendors' tools. Certainly, the model will contain common core functions; also, it will allow facilities for vendors to extend the information model with key features that uniquely identify their tools. It might also be observed that some of this proprietary data may not always be stored in an information model. To accommodate some of these tools, a repository must also allow the use

[1]Open Information Model (OIM)

of either references to external files, or the storage of binary data streams, Binary Large Objects (BLOBS), within the repository database. This further highlights the need for an information model to be dynamically extensible.

2.3.4 Generic Repository Tools. Though this fourth aspect of a repository is not described explicitly as a part of the architecture by Bernstein [6,7,9], a repository is of no use without it. Functions for searching, browsing, import and export, and scripting are utilities that provide the capabilities needed by a repository user.

Scripting allows users to generate desired data automatically from information in the repository. For example, an entity-relationship diagram tool might have a scripting language to generate automatically DDL, DML or even SQL query interfaces from the diagrams of the tool. In a data warehouse tool, a graphical representation of the mappings between the external data and the warehouse schema might use scripting languages to generate an Extensible Markup Language (XML) translation between the two formats [8].

Import and export functionality is necessary to add previously existing artifacts to the repository. Meta-data representing repository artifacts must be shared between tools. These tools may be implemented on top of separate models (if they were pre-existing in another repository, for example). Likewise, a repository should offer a means to export its meta-data to another tool.

Software artifacts progress through phases during development and maintenance—the software lifecycle. Repositories should offer a workflow control model to track the artifacts. Meta-data about the workflow control process can be stored in the repository. This meta-data will describe how products flow through the process. It will allow tools to

manipulate an artifact only when the process dictates. A workflow model will indicate the order in which tools are used in the process and can easily be updated as new tools are introduced or improved processes are proposed.

Several tools are used in the AFIT software synthesis process. This research explores how repository technology can be used to enhance this process and solve some existing software synthesis problems. The current AFIT software synthesis process is described next.

2.4 Software Synthesis

AFIT KBSE uses software transformation to synthesize new programs from existing domain knowledge. A software transformation system (mostly) automates the transformation from a formal specification to a design specification. The main idea is to reuse existing domain knowledge to generate requirements specifications. The requirements specification is maintained—not the source code. From these formal requirements specifications designs and source code are "system" generated. As stated in the last chapter, AFIT research has resulted in software synthesis tools, generic models for object-oriented domain theories and designs, and a generic model for legacy software coded in imperative languages. Figure 1 (Chapter I) showed the software synthesis tool process flow.

According to Smith [45] software synthesis is defined as a five step process starting with the development of a domain theory for the problem to be solved. The process then proceeds by creating a specification describing the problem using the domain theory, and refining the specification into a program-based model. Next is the application of program optimizations. The process ends with program compilation.

Just like any other software engineering process, the products of the KBSE will also have constant maintenance requirements. As a result, traditional versioning and configuration management schemes apply to the KBSE environment. As requirements change, it becomes necessary to generate new versions of artifacts. Additionally, one may desire alternative solutions for the same requirement. For example, a software designer may consider alternative designs for a particular target system. The software synthesis tool would generate two versions of the target program based on the design transformation options chosen. The designer may then evaluate performance of those two design options against actual end-user data to determine which design decision might offer the best performance. Ideally, the modified data will be saved as a new version, and the old data will be retained.

In addition to automating the development process, a KBSE environment should automate the maintenance process. This can be accomplished by retaining enough information about the process of synthesizing the software designs to be able to "replay" the development [3] to recreate the design automatically once modifications have been made to the formal requirements specification. This allows the maintenance of requirements rather than code.

2.4.1 Information Model for Software Synthesis. AFIT has extended and enhanced this process to include the development of a library of reusable object-oriented domain theories. These domain theories can be specified in a formal language that has been extended with object-oriented concepts [25]. Alternatively, they can be specified using informal, graphical techniques extended with formal semantics [16, 36].

AFIT researchers have developed separate object models to describe domain knowledge, and generic object-oriented programs. Research has produced a software synthesis tool capable of limited transformation of formally-specified domain knowledge into a software system [26].

The current environment contains a number of different, but similar information models (or tool-specific object models). Two separate models, the Generic Object Model (GOM) and the Generic Imperative Model (GIM), were both originally developed by Sward [46]. The GOM is a generic representation of an object-oriented programming language in an abstract syntax tree (AST). This model has been used as the design model for AFIT software synthesis. The GIM is a generic representation of an imperative programming language in the form of an abstract syntax tree. The GIM's predominant purpose is to facilitate the reengineering of an imperative program into an object-oriented program by parsing an imperative language into a GIM AST, then transforming from the GIM into the GOM AST. Using these models, researchers were able to successfully semi-automate the transformation of legacy Fortran [46] and COBOL [15,35] programs to an object-oriented design AST.

In addition to the GIM and GOM, AFIT researchers have developed a generic model for representing domain knowledge [16, 26]. This domain model (DOM) allows formally specified domain knowledge to be represented and reused. These three separate models, in addition to language-specific models for the input language, Z, and specific target languages have been developed.

AFIT research could use seven to ten separate object models, depending upon the number of target languages, to enable a single main task—the transformation of domain knowledge into, ultimately, source code. The current software synthesis tool suite uses four separate models. In the next section this document reviews these tools.

2.4.2 Software Synthesis Tools. AFIT has developed tools for refining domain theories into formal requirements specifications [1]. This tool, called Elicitor-Harvestor, uses AI techniques to assist a user in understanding, refining and augmenting a particular domain theory into a requirements specification. Both domain theories and requirements specifications are represented in the DOM. Additionally, the knowledge used to transform domain theories into requirements specifications must be maintained. Currently the tool used to create these requirements specifications retains the sequence of operations applied during transformation. This data is ultimately what associates domain theories with requirements specifications. The most current version of the tool accepts a single domain theory as input.

The surface syntax for a domain theory is currently based on the formal language, Z, using the typesetting language, LaTeX. A tool developed at AFIT parses the LaTeX-specified domain theory into the DOM. More recently Noe [36] extended Rational Rose to generate the LaTeX-Z surface syntax from graphical Unified Modeling Language (UML) specifications. The parser then builds a DOM AST from the Noe output.

AFIT KBSE tools operate primarily upon single instances of ASTs. In general, the exception is tools that make the transformation from the DOM to the GOM. In addition, Elicitor-Harvestor copies the instance of the domain theory AST to a duplicate instance

internally. The tool uses this method to preserve the structure of the original AST while it is being refined into a requirements specification. Transformations take place in the "copy." There is currently a methodology for merging multiple domain theories into a single AST via the AFITTool parser. This merger is completed by matching names (class, attributes, etc). No mechanism, or supporting meta-data is provided to aid in verification of the domain theory merger—it is up to the user to know the domain theories fit together.

Once a formal requirements specification is complete, it can be transformed into a design using automated/semi-automated transformations. These transformations are applied with assistance from a software engineer. Currently, AFIT researchers have developed tools that generate source code for primitive and aggregate objects [32, 47]. Current research is underway to transform the dynamic model to code [33].

Additional tools and methodology have been developed allow the use of other popular CASE tools in this process [36]. All of the data related to these tools is stored in a variety of data structures, in a number of locations in a hierarchical file structure. Other than this file structure and an associated catalog listing the files and their "part numbers", no environment, descriptive meta-data, or reuse-oriented search capability is available to ensure this data can be easily located, managed, and shared among the tools.

In addition to a reuse-oriented environment, there is currently no way to represent a key concept of software synthesis, the ability to replay designs when there is an inevitable requirements change. Since part of this research will propose a methodology for creating a design history, the next section discusses the composition of software designs.

2.5 Software Designs

The early work on software development completed by software engineering pioneers such as Dijkstra, Wirth, and Parnas introduced the concepts of "stepwise refinement" [14,39,51]. In these works, the authors describe software development as a series of step in which a developer begins with an intermediate stage and proceeds with design decisions. Any design decisions made prior to a selected intermediate step will be shared by all of its descendants.

In stepwise refinement, Dijkstra referred to programs that are complete except for the implementation details of certain operations or types. Also, this work introduces the concept that all designs are a tree (Figure 7) of intermediate designs connected by "abstract decisions."

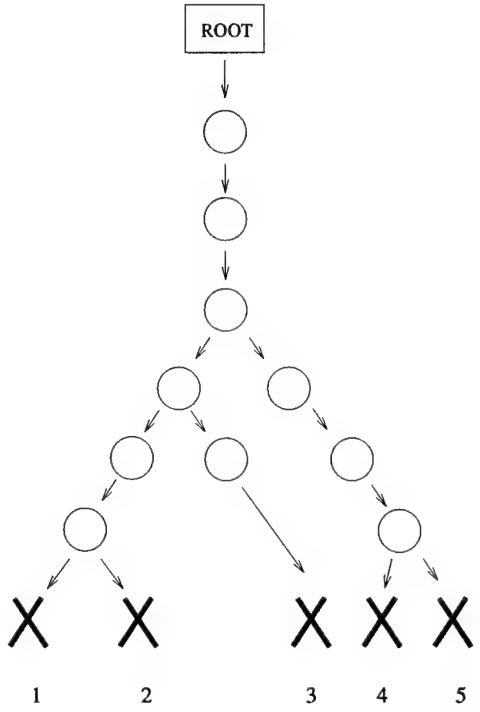

Figure 7. A sample design tree [39]

Parnas further showed the designs, which are descendants of a single parent, form what he calls "program families." He suggested that these program families will consist of many versions due to necessary improvements[2].

Frakes [20] looked at program families in his work on software classification. Frakes stated that a potential problem with the concept of program families is the amount of design information that must be captured during development of a large system.

Such a system would require all the alternatives to be known at the design stage—not always fully possible during a manual design process. For example, a developer may complete a program according to its design, but find the that a certain, chosen algorithm did not perform as expected. Frakes said it is equally difficult for a (human) developer to document each and every design choice, or that the design choices could be presented in a way they can be used.

2.6 Summary

This chapter showed that the underlying technology of a repository is a database management system (after all, database management systems exist to *manage data!*). Previous efforts investigated for this research have used either an OODBMS or a relational database with an object-oriented layer (built into the repository engine). Each repository has differing philosophies, along with some similarities, when representing relationships in the repository object model.

[2]Multiple experimental versions, or versions for multiple platforms could also be present.

The chapter also established background in software synthesis–particularly AFIT's software synthesis environment. The many kinds of data, various models, and existing tools were discussed. Perhaps the exploration of repository technology can be an effective solution to the highlighted short-comings of the current software synthesis tools.

The repository information model must allow the various software synthesis tools to share appropriate data. Additionally, the tool must allow meta-data that facilitates the identification of the appropriate reusable artifacts. Finally, the environment should allow any automated transformations within the system to be kept, at least temporarily, until the new software system is synthesized.

Once all the transformations are complete, the correct sequence of transformations and any meta-data used in the derivation of the new system will then become an additional set of meta-data that represents another software synthesis artifact—a design history. This "artifact" should also be stored in the repository. The next chapter will address specifically the requirements and methodologies for using a repository for software synthesis.

III. Requirements and Methodology

For a repository to be useful in a software synthesis environment, the various concepts of software synthesis must be adapted to repository technology. For example, a repository-based process must be adopted, an appropriate information model for software synthesis must be developed, and artifacts of the software synthesis process must be identified. Also, repository relationships must be identified and modeled to aid in selection, understanding and reuse of these artifacts. Finally, using the repository's capability to represent relationships between various artifacts, a solution to a long-standing software-synthesis problem, design histories, can be suggested.

This chapter explains the methodology used to adapt software synthesis to a repository environment. It also proposes a repository relationship-based solution for recording and replaying design histories. The following sections provide greater depth on the requirements identified and offer the methodology this research uses to solve them.

3.1 Repository-Based Development

The process shown in Section 1.1, though adequate to fulfill the task of software transformation, does not provide a sophisticated representation or searching capability reuse experts suggest are needed to efficiently retrieve existing artifacts such as domain knowledge. Search tools must help a user find and select appropriate artifacts for reuse in a new system (screening, identification, and decision from Section 2.2.3.2). Additionally, users in a large development environment will not follow the serial process proposed previ-

ously. Instead, many parallel activities will be ongoing. For the software synthesis system to become viable, an appropriate process and reuse infrastructure must be in place.

3.1.1 An Infrastructure to Support Reuse. The previous chapter showed that experts often present repository-based systems as a key technology in meeting the needs of component reuse [7, 13]. For source code, repositories are successfully used to achieve software reusability. The repositories capture meta-data about source code and can provide powerful search tools. The meta-data furnishes appropriate abstractions, organization, and management capabilities to promote reuse. Appropriate search tools use this data to communicate whether the functionality users seek is available in a component in the repository [13]. In this repository reuse process the various tools used in the development process are all clients of the repository. Common functions needed by the client tools will be provided by the repository.

Recall the various tools and data involved in software synthesis. First, Elicitor-Harvester assists a user in selecting and refining formal domain knowledge into a formal requirements specification. This tool uses meta-data, such as a thesaurus-like function. It also uses additional data to keep track of the sequence of transformations used to refine a domain theory into a specification. The domain theory and specification round out the data read or created by this tool.

Once the requirements specification is complete, the design tools take over. They refine the requirements specification into a design using successive design decisions applied by a software engineer. Not only the final design, but also the design decisions (and their rationale for selection) are ideally part of the data to be retained in the environment.

From the design specification, the source code can be generated automatically by a formal translation from the design representation. Finally, as discussed previously, other CASE tools can be used in the process. Their persistent data must also be retained for reuse.

All of this data has potential for reuse in future development and maintenance. That is the reason the software synthesis process in Figure 1 showed the various "libraries." Currently, the libraries exist simply as files and directories in a Unix file system. This library data will be more appropriately managed by employing the same repository technology used historically to manage source code. The data will be stored as artifacts in the repository, and the software synthesis tools will become clients of the repository (Figure 8).

There are a number of other areas where the AFIT software synthesis environment can benefit from repository technology. Some of these areas are known shortfalls in AFIT-Tool, such as the development of design histories. Other areas have been solved using tools-specific approaches, but could be made widely available to all tools in the environment if implemented as a standard set of functions within a repository engine (recall Section 2.3.1). This might include Elicitor-Harvestor functions for searching and understanding domain theories [1].

3.1.2 A Reuse-based Process for Software Synthesis. The software synthesis tool process flow, in this environment, will no longer appear as the serial one traditionally proposed—a kind of "batch sequential" process. In the traditional process tasks and tools are introduced consecutively and the output of one tool is the input to the next. Instead, this research proposes a repository-based process in which multiple developers use the various client tools to locate appropriate reusable domain theories, domain experts add

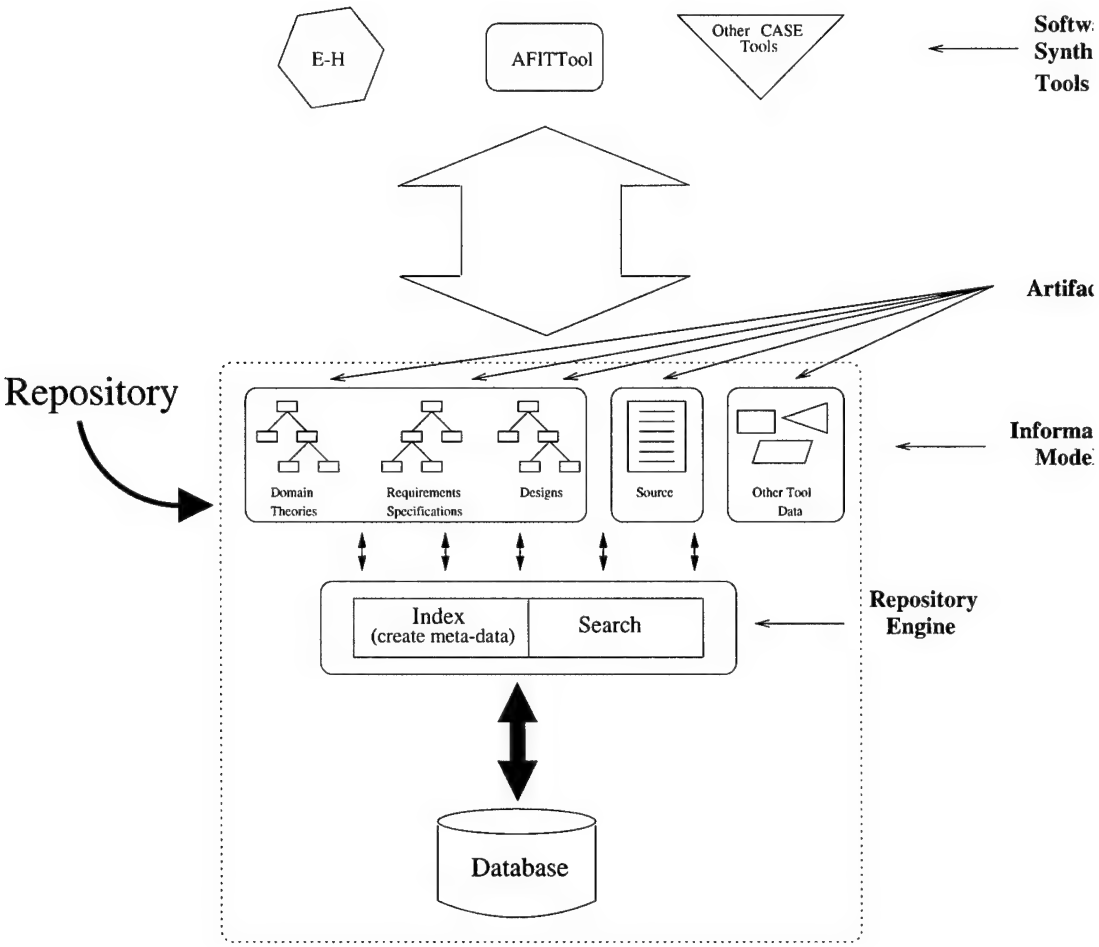

Figure 8. Repository-based reuse (adapted from [7,13])

or modify domain theories, still other developers refine domain theories into requirements specifications, and software engineers transform requirements specifications into design specifications and generate code from existing designs.

The repository database provides much of the functionality necessary for the concurrent, multi-user environment described—including locking, data integrity functions, and backup and recovery. For example, the DBMS will ensure the ACID properties (atomicity, consistency, isolation, and durability) for each transaction are satisfied. Rudimentary functions for object versions are also available from the DBMS. Beyond that, functions such

as version management and configuration management will be available via the repository engine.

With the powerful capabilities provided by a repository, the software synthesis tools may be used in any order deemed necessary by software synthesis experts. For example, already some AFIT researchers have suggested inserting software architecture tools in different points of the process [36,50]. The repository can easily facilitate this by allowing the introduction of the workflow model concept. New tools can be introduced without modifying old tools or data. The workflow model as applied to the current software synthesis process is shown in Figure 9.

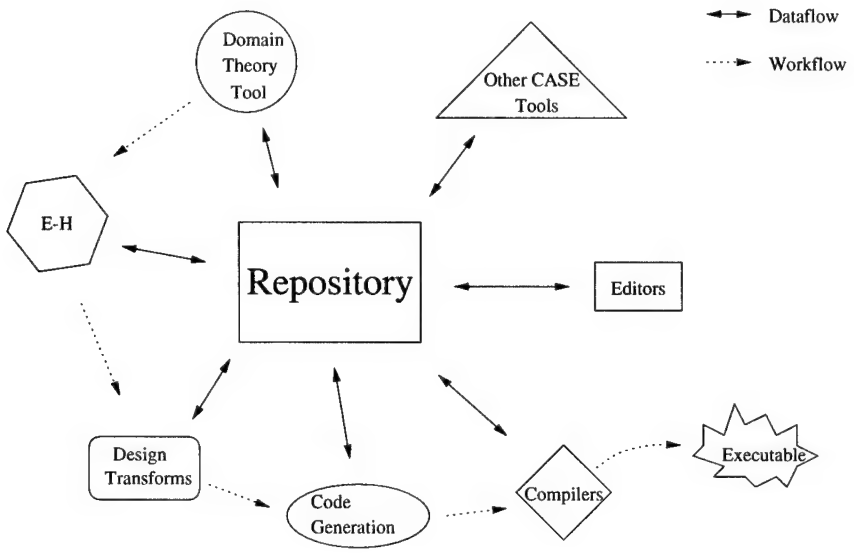

Figure 9. Repository-based software synthesis process

A key ingredient for allowing a process to arbitrarily introduce new tools or artifacts is to share object models. This shared object model, the repository's information model, is the subject of the next section.

3.2 Shared Information Model for Software Synthesis

A shared information model for a software synthesis system must represent the semantics of all the client software synthesis tools that will share the model. One way to find a common representation of the semantic concepts of the multiple, current models used in AFIT software synthesis tools is to merge the existing models into a single model. This new information model provides a common data format on which all of the software synthesis tools can operate.

3.2.1 Methodology for creating a common model. During this effort, Graham developed a new language model, the Common Object and Imperative Language (COIL) [24]. The COIL combines and refines the GOM and GIM into a common model. A major goal of the COIL was to capture common core features of object and imperative languages, but minimize "convenience features and shortcuts" to keep the language simple [24].

This new model simplifies the task of reengineering because there are no longer two separate object models for abstract syntax tree instances. In the new model, various identifiers from the imperative program, such as variable and type names, can be retained in the object-oriented version without transformation. In the previous paradigm, using the two models, information duplicated in each model would have to "transformed" (copied) from the GIM to the GOM.

In addition to the GIM and GOM, the current process uses a separate model (the DOM) for the representation of domain knowledge and requirements specifications. This research developed a single model capable of being shared by all the major tools involved

in the software synthesis process. Since the desire of this research was to merge the various software synthesis models, a suitable methodology for completing the process was created.

Existing methodologies for merging object models could have been used. For example, a methodology developed in previous AFIT research [2, 12] merged complex object models for simulation systems. However, a simpler methodology for merging language models can be inferred from Graham's presentation of the new COIL model [23]. The concept and results can be analyzed to extract a high-level methodology for combining two language models. The methodology is described below.

1. *Select one of the models as a basis.*

   The other model is "added" later. In the creation of the COIL, Graham first started with the GIM. In this first step, he carefully captured the semantics of the GIM in the COIL. Object-oriented concepts of the GOM were then added.

2. *Identify semantically similar elements of the two language models.*

   Both the GIM and GOM include basic semantic concepts such as type definitions, variable declarations, statements, and binary expressions (equal, less than, and, or, add, subtract, ...).

   (a) *If a semantically equivalent element is already represented in the selected base model, no changes are necessary.*

   (b) *If an element in the external language is not in the base model, determine if it can be incorporated into an existing syntax in the base language.*

Graham added the GOM to the initial "imperative" version of the COIL extending several elements. For example, an object-oriented class was considered to be another element of the COIL's type system. Similarly, methods were simply considered to be subprograms with the additional semantic of being a component of a class.

3. *If there is no semantically equivalent element in the base language, it should be added.*

   The GOM implementation of the object-oriented concepts polymorphism and inheritance were added to the original "imperative" COIL. For example, an additional attribute was added to the class to hold superclass information.

This methodology was used to combine the COIL and the DOM. The COIL was selected as the base language. The application of this methodology to compose the new model, the AFIT Wide Spectrum Object Modeling Environment (AWSOME)[1] will be discussed in Chapter IV. Major components of the AWSOME model are described in Appendix B. Once this single model was in place, we considered the artifacts to represent in this model, and determined the relationships between them.

3.3 Software Synthesis Artifacts

In software synthesis, artifacts must be thought of differently than in the traditional software process. Current reuse systems focus on the understanding, selection, and reuse of source code objects from the repository. Since KBSE focuses on reuse of knowledge, the domain theories will be artifacts of primary interest to reuse tools—those tools designed for

[1]Name attributed to the head of the KBSE Research Group, Dr. Hartrum.

the selection and understanding of reusable artifacts. Additionally, many other products of the process might be considered artifacts. With a common model, software synthesis artifacts may be represented in a single model, and perhaps in the same AST. This section discusses what this research defines as the artifacts of software synthesis and how the artifacts were chosen.

3.3.1 Artifacts. The list and number of artifacts from just the basic process proposed in previous software synthesis research [1, 26, 32, 36, 47] is not trivial. First there is the range of domain theories that must be developed and represented. Additionally, the Elicitor-Harvester tool [1] produces a formal requirements specification. The requirements specification is transformed automatically into a design. The collection of automated transformations provides the design history of the software synthesis process. These design histories as well as the designs produced must also be retained in the repository. This provides reusable designs. From them, users can generate multiple versions of a system in different target languages or, perhaps, for different computing platforms. This source code is also another artifact that might be represented and retained in the repository. The basic set of artifacts involved in the AFIT software synthesis process is summarized in Table 4. Additional proposed software synthesis artifacts are in Table 5.

In addition to these software synthesis artifacts, data from other CASE tools can be used. Already, AFIT researchers have formally specified requirements using informal graphical modeling tools as part of the software synthesis process [16, 36]. These tools

Table 4. Chief artifacts of the AFIT software synthesis process

Domain Theory	A formally represented set of knowledge of a given domain.
Requirements Transformations	Activities taken by a user during the refinement of domain theories into a requirements specification.
Requirements Specification	A formally specified set of requirements derived from one or more domain theories.
Design History	The recorded exploration of the design space as rules are applied to requirements specifications to transform them to designs.
Design	All requirements specifications are transformed to a generic object-oriented design model.
Source	Designs are transformed into source code–either by direct processing of the generic model, or by transforming a design into a language-specific abstract syntax tree.

Table 5. Some other software synthesis artifacts

Algorithms	Formally expressed, common algorithmic constructs (e.g. Divide and Conquer, Global Search) [45].
Architectures	Architectural blueprints for connecting system components (e.g. Pipe and Filter, Blackboard, Client-Server) [50].
Design Patterns	Common solutions to recurring problems in a given domain [22].

will store their data in a common tool data model (e.g. UML), as path references to tool-specific data, or as binaries retained in a tool-specific format[2].

With the exception of text of the source code, these AFIT KBSE artifacts will be represented in the common information model. Transformations can then be applied to the common model. Each transformation will change the state of the AST to which it is applied. Some states will be intermediate, transitional states (as suggested in Section 2.5). Other states will be complete artifacts. In fact, depending on tool implementation, a single AST may consist of all the meta-data necessary to represent more than one artifact (see Section 3.5.3). In addition, other artifacts such as proprietary tool data or source code will be represented in other information models, proprietary binary, or text formats.

[2]In database terminology these are known as Binary Large OBjectS (or BLOBS)

It is also important to note that every artifact can have description information. This description will provide the most value to information retrieval tools and techniques that might be employed on this repository. Figure 10 shows that an artifact will have an attribute to record its role (e.g. domain theory, requirements specification, etc) and another used to indicate the format of the artifact (e.g. AWSOME, text, Word6.0, etc).

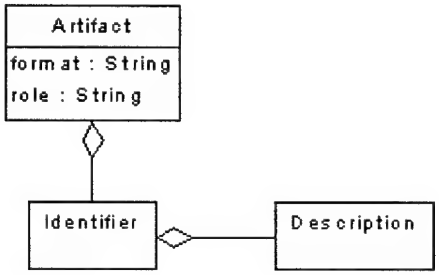

Figure 10. The composition of an artifact

3.3.2 Software Synthesis Relationships. The primary artifacts of software synthesis will be related within a repository. For example, Anderson [1] described a formal requirements specification as a refinement of a single domain theory. That is, a requirements specification is simply a "pared down" domain theory,[3] suggesting a one-to-many relationship between a domain theory and requirements specifications. However, it is easy to imagine the participation of multiple domain theories in the development of a single requirements specification. For example, separate "Person" and "School" domains might be developed, then may be combined in a requirements specification for a "University System" where many elements of the Person domain are used to complete the full specification

[3]A user may also enhance the requirements specification with new information not modeled in the domain theory.

of students and instructors in the School domain. The example suggests a many-to-many relationship between requirements specifications and domain theories.

Also, a software engineer may develop more than one design for a given requirements specification. For example, a software engineer may develop one solution that implements an association as a one-way pointer and another design solution that implements it as a two-way pointer. Each of these design alternatives would be derived from the same requirements specification. Therefore there is a one-to-many relationship between requirements specifications and design alternatives. Each design alternative can then be automatically transformed into source code. The source code might be generated in several target languages, also a one-to-many relationship.

Anderson [1] describes the relationship between a domain theory and requirements specifications as a sequence of operations applied to transform the domain theory into a requirements specification. Similarly some sequence of transformations will change a requirements specification into a particular design. The relationship between the main software synthesis artifacts, as described above are shown in Figure 11.

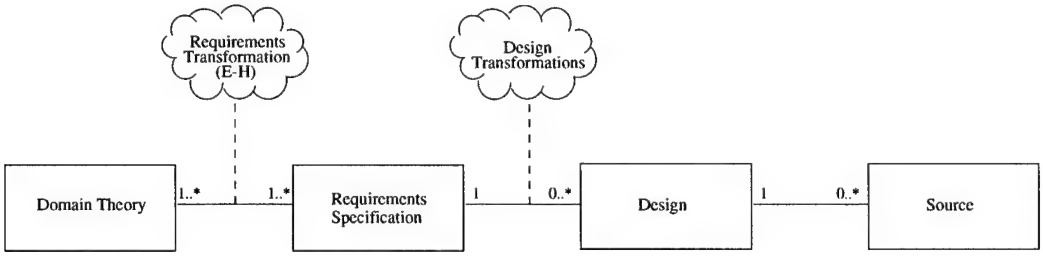

Figure 11. Relationships Between Artifacts

3.4 Repository Meta-Model

It is necessary to provide the capability to model the relationships described above. This research used previous work in [13] to develop generic methodology for capturing relationships.

3.4.1 Application Frames. In [13], the authors develop a model in which artifacts are collected into application frames (Section 2.3.1.1). In this research, a similar methodology will be used to relate artifacts. Instead of an application frame containing one mandatory implementation, the minimum configuration of a software synthesis application frame will contain one mandatory requirements specification and a mandatory reference to the one (or more) domain theory(ies) from which it was derived (Figure 12).

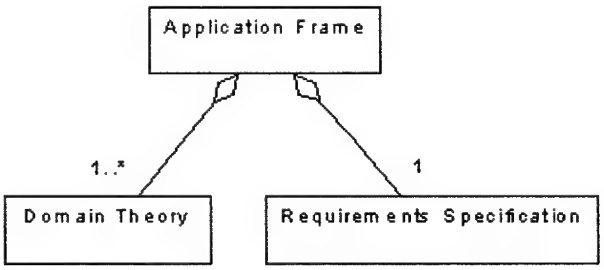

Figure 12. An application frame

An example application frame may consist of more than the minimally configured application frame; however, it might also contain a number of other artifacts. For example, a particular design artifact might be related to a specific source code artifact, as well as a DDL artifact defining the application's database. Another design alternative might use an object-oriented database in which a slightly different design and source code artifact will be related (no DDL required). These example design alternatives are shown in Figure 13. An application frame containing these two design alternatives is shown in Figure 14.

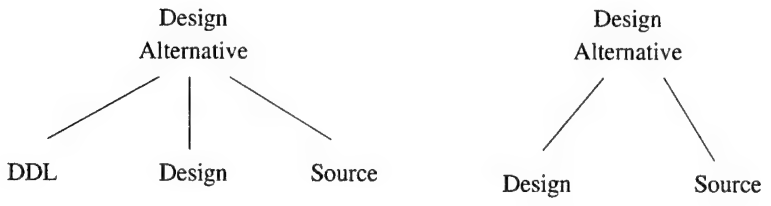

Figure 13. Related design alternatives

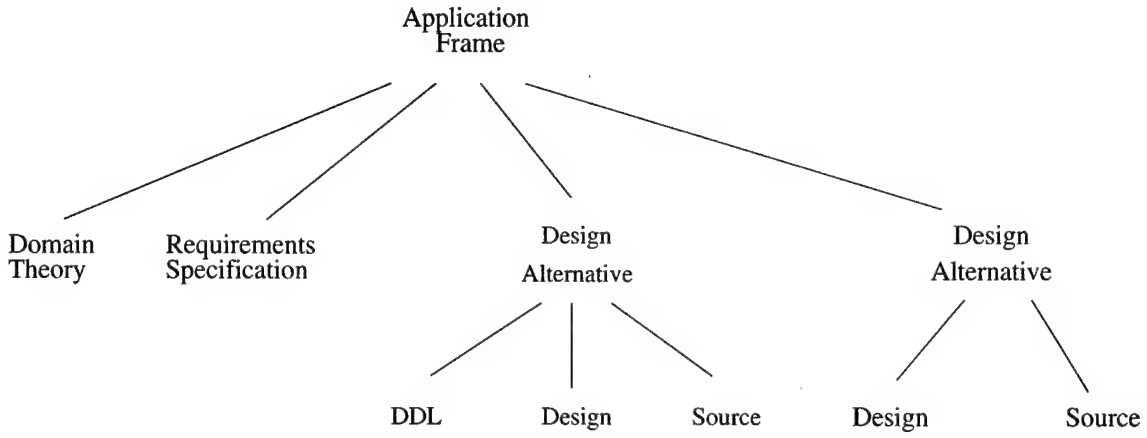

Figure 14. An example application frame

From these examples, a methodology for describing an application frame, in general, is taking shape. An application frame will consist of a single requirements specification, one or more domain theories involved in its creation, and a number of design alternatives (Figure 15)[4]. The general methodology used for application frames can be adapted to the representation of other kinds of repository relationships.

3.4.2 Repository Relationships. Now that the application frame from [13] has been adapted to an application frame for AFIT software synthesis, this research described a generalization of the application frame and used it to model other relationships in a repository. The general repository relationship is described in Figure 16.

[4]For the moment, the history artifacts are omitted and are addressed in a later section.

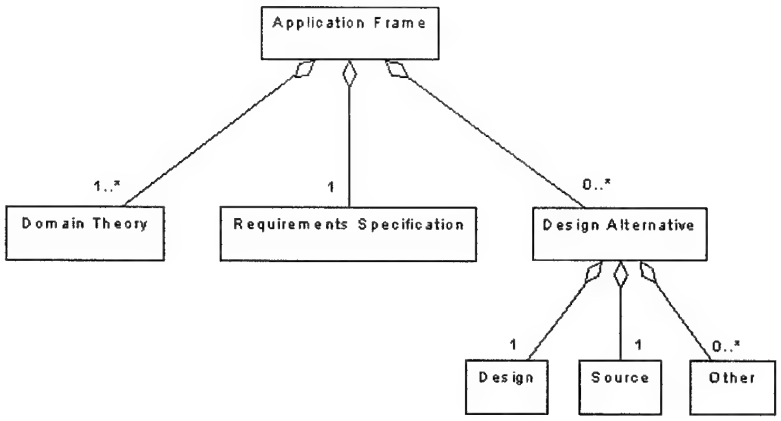

Figure 15. Model of an application frame

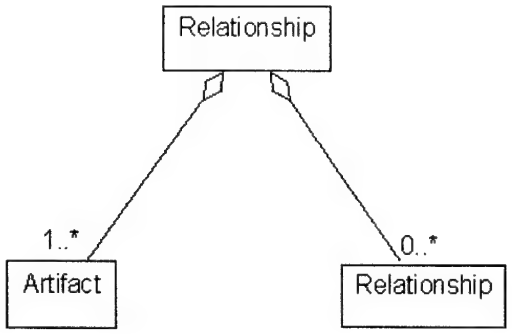

Figure 16. General model for repository relationships

With this methodology, the repository can describe many arbitrary repository relationships, in addition to the application frame previously discussed. These relationships include repository engine functions such as similarity, version, and configuration.

3.4.2.1 Similarity. Instances of similarity relationships can exist for each collection of "similar" artifacts. As in any other case, a single artifact instance will participate in potentially many similarity relationships. For example, any domain theory that uses the simulated Fuel Tank domain, such as an aircraft, might participate in similarity relationships. Likewise, simulation aircraft in the various aircraft domains might participate

in a similarity relationshipIn the fuel tank example each aircraft domain will participate in both aircraft and fuel tank similarity relationships.

3.4.2.2 Version. Version relationships can be defined in one of two ways. A more simplistic view of versions might describe versions of artifacts as a sequence of artifacts in which artifacts derived from a common ancestor are related temporally by sequence position (Figure 17).

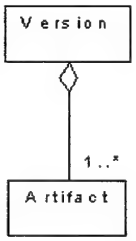

Figure 17. Oversimplified version relationship

A more complex methodology for representing version relationships in the repository will also allow successor (and implicit predecessor) relationships. However, versions will be related in a way that allows multiple versions to have common ancestors, and will allow later versions to be merged (Figure 18). The more complex version model will allow more powerful semantics in describing relationships between versions. Though these versions are obviously associations in object-oriented terms, the methodology shown here is based on an aggregation paradigm that allows the abstract relationship proposed in this research to hold. This also allows this representation to be consistent with the Parnas discussion of design versions mentioned earlier (Section 2.5).

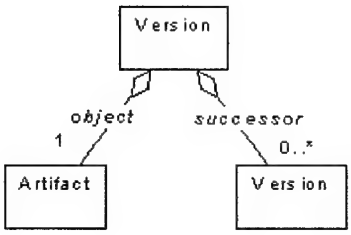

Figure 18. More powerful version relationships

3.4.2.3 Configuration. A configuration is made up of versioned instances of artifacts along with other, possibly nested, configurations. The reader will recognize this is very similar to the application frame constructed previously. In fact, a configuration *is* an application frame with the concept of version introduced. Configurations are, themselves, versioned.

The final relationship discussed is the design history. Before describing the history relationship, we develop a methodology for describing design histories.

3.5 Design History

When a software synthesis system generates automated designs, it is up to the user to explore the "design space." A design history represents the steps, in the form of transformations, a user takes while refining a formal requirements specification into a synthesized design. The complete *design space* can be thought of as all the possible paths that can be achieved by applying various design decisions in the form of transformations. As can be imagined, implementing a fully automated exploration of the design space is combinatorially explosive and falls into the realm of NP-hard problems.

To overcome this computational difficulty, a user must control the application of design transformations. That is, rather than the system trying all combinations, the user will attempt to apply appropriate transformations—those that progress toward a valid design solution. However, the fallible user will still select some inappropriate design transformations and apply poor design decisions.

Any model intended to record a design history must be capable of allowing the user to recover from these bad choices. Obviously, the sequence of transitions that leads to a valid solution must be retained. However, the user may also want to retain partially explored solutions, and mark bad transformations and design choices so the same mistake won't be repeated in future exploration of alternate designs.

Finally, the user may desire to retain a variety of deliberately created alternatives. For example, design alternatives representing the previously mentioned one-way versus two-way pointer might be explored. Additionally, alternate software architectures (e.g. pipe-and-filter, layered, or object-oriented) might be investigated.

3.5.1 A Notional Example. Perhaps a simple, generic example can best illustrate the user's exploration of a design space. Figure 19 shows a design space starting with a formally specified requirements specification. In this example, suppose at least three transformations using the initial requirements specification as input are possible. Before explaining this example, it is important to note that the exploration of the design space can occur in any order, not just the level-by-level method shown.

From here, the user explores three more design transformations from the first result of the previous step (design 1.1). Additionally, the user explores two possible design

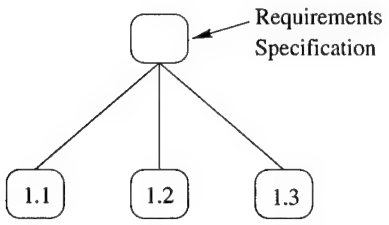

Figure 19. First level of transformations

transformations from the third result (design 1.3). The user chooses not to explore any transformations from the middle result of the previous step. Figure 20 shows the design history to this point. The middle result may have further valid transformations, but this is unknown, since the user chose not to explore them.

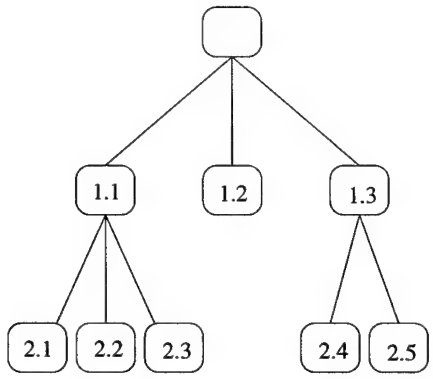

Figure 20. Second level of transformations

At the next level of the design space, the user (or system) may determine transformations beyond 2.1 are not a valid solution and no possible transforms along this path are possible. Nodes 2.3, 2.4, and 2.5 may or may not have further possible transformations, but the user decides not to explore these possibilities at this time. The user chooses to explore possible designs that flow from node 2.2.

At the third level, the system lets the user know 3.3 that no further transformations are possible from that node. This process continues until the user completes the design.

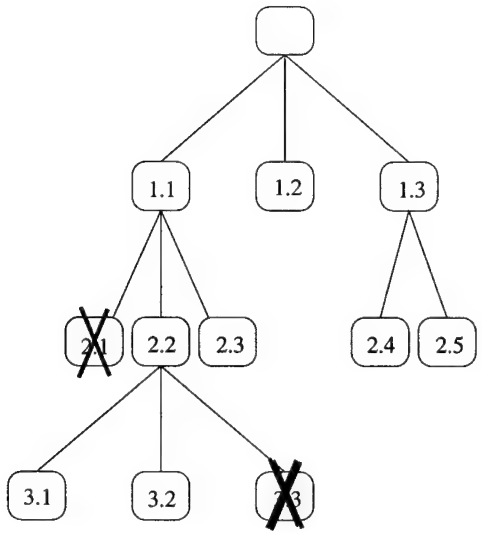

Figure 21. Third level of transformations

What remains is a "meta-tree" of the design space explored. Each arc of the design space meta-tree represents the transformation applied. Each of its nodes is the design tree resulting from the previous transformation.

3.5.2 Node Types. Each node in the design space tree fits into one of four categories: complete, viable designs; non-viable designs; partially-explored designs; and unexplored designs. It is important to distinguish between these design alternatives because partially-explored and unexplored nodes may have children that lead to other complete, viable design alternatives.

3.5.2.1 Partially Explored. In a partially-explored node, the first transform may lead to a partial design from which a transform was successfully performed. However, it is possible other transformations could be explored from this node. Partially explored nodes are important because they may represent a path toward another design alternative.

3.5.2.2 Complete-Viable. A complete, viable design is a valid solution to a requirements specification. It will have no transformation arcs exiting from it. The sequence of transformations leading from the requirements specification to this node in the design meta-tree represents the design history of this design alternative.

3.5.2.3 Non-Viable. A non-viable design alternative is one that "bottoms out" before realizing a viable solution. It may either return to a previous state, be subjectively declared a bad alternative, or will simply transform no further (hopefully, reporting an error). It will also have transformations exiting it, but they will not have a design AST as output because the transformation tool forbade the transformation or reported an error.

3.5.2.4 Unexplored. Finally, there is the unexplored design alternative. Unexplored design alternatives will have no transformations leaving them. The user simply gave up further exploration of this path, then decided to take a different route (or perhaps simply stopped here to perform a different task). There may be some overlap between unexplored and non-viable alternatives[5].

3.5.3 Summary. Figure 22 shows how these four alternatives (complete, viable designs; non-viable designs; partially-explored designs; and unexplored designs) apply to a design meta-tree based on the notional example given previously. As can be seen in the diagram, any completed design has a single path of transformations leading from the root requirements specification to the design. Any node between the requirements specification and design will represent a hybrid of requirements and design information. Because of

[5]Since this process is controlled by a user, the decision that one alternative is bad is subjective—one man's trash is another man's treasure!

the wide-spectrum model proposed during this research, a client tool implementation may retain all the requirements information within a particular design.

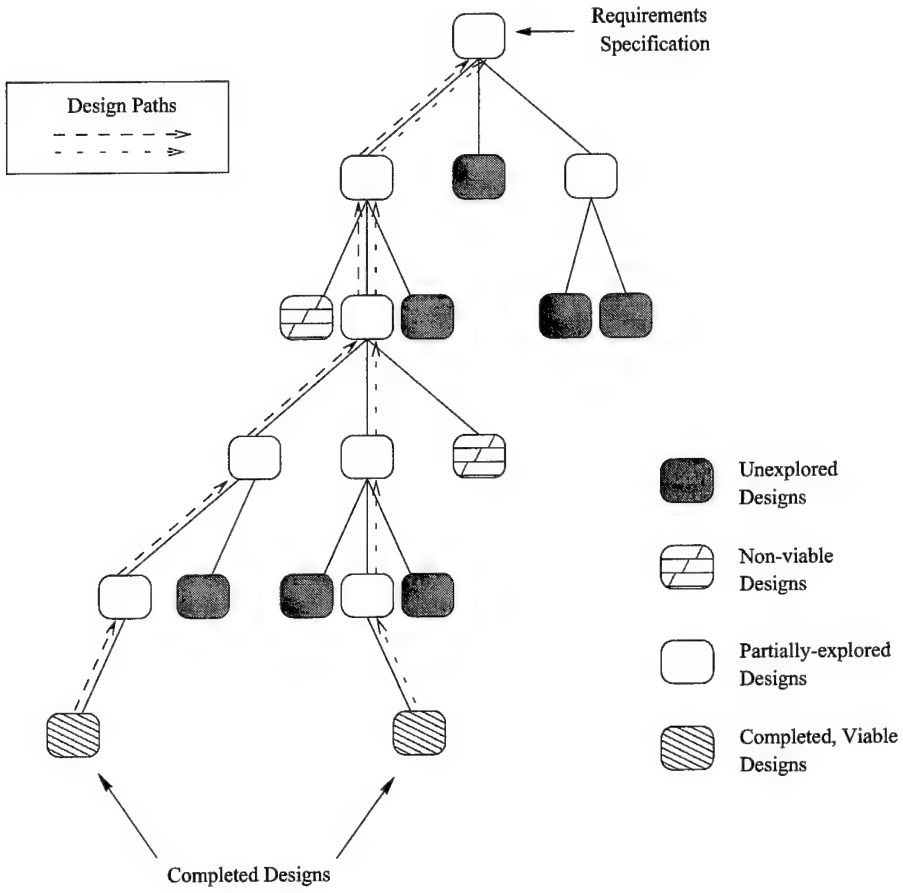

Figure 22. Four categories of design alternatives

3.6 Methodology for Representing Design Histories Using Repository Relationships

Using the relationships introduced in Section 3.4, it can be shown how a repository can provide users the capability to represent design history information. The technique is based on the philosophy that each design alternative is related to a previous design alternative by exactly one transformation.

The relationship that allows the building of an arbitrary tree similar to the one described in the previous section is depicted in Figure 23. Note the similarity to this methodology for representing versions. In fact, this methodology maps very nicely to the version representation described in this research. The transformation applied will be retained in the meta-data of the relationship.

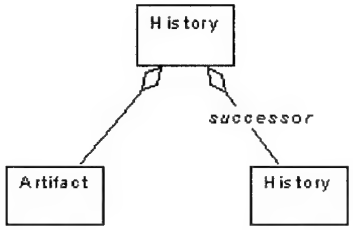

Figure 23. A relationship model for design history meta-trees

Once the client tools assist the user in developing a completed design, it may not be desirable to save the entire meta-tree. For purposes of this research we have ignored the heavy space requirement of the "tree of trees" methodology. Future research can apply typical, space saving techniques such as delta-versions (saving only successive changes to the original). The system should automatically save the sequence of transformations involved in a completed design. This is done by simply traversing the meta-tree from the completed design leaf node back to the root requirements specification. The user may want to optionally save and document unfruitful choices or uncompleted designs. This model will allow this to occur by allowing a client tool to prune the meta-tree, as necessary. The tool designer must be careful to reconnect any desired relationships that were disconnected during the pruning process.

Any stored design rationale and especially negative results from pruned nodes should be propagated back to the highest level design node retained in the design meta-tree[6]. This will allow the tree to keep information about any successive "bad" transformations. This kind of information can prevent future users from pursuing a previously considered alternative design that will "bottom out" several levels down or results in a program with poor performance.

3.7 Summary

In this chapter, a new repository-based process and software synthesis object model were proposed. The repository will allow shared data, and the modeling of the many complex relationships between the software synthesis artifacts. It also showed the methodology used to represent design histories within the repository.

The next chapter will discuss the creation of the new information model, and will show specific examples of the model. It also shows the successful application of our methodology for merging language models. The development of the repository and design histories is discussed in Chapter V.

[6]This is akin to an A* tree search algorithm. A* marks the top node of a sub-tree with the best solution within it so later traversals will not reexamine the sub-tree. This can effectively prune bad solutions.

IV. A Software Synthesis Model

The top level in a repository architecture is the information model. The development of an information model for software synthesis, therefore, became the first task of this research. As stated in Section 3.2, the COIL and DOM were integrated into a common model using a methodology demonstrated during the development of the COIL. The COIL was selected as the base model. The DOM was the external model to be "added" to create the new, integrated model.

4.1 Merging Models

The first step of the methodology was to identify common elements. In the creation of the AWSOME model, the most logical place to start was the top-level object of each model—the root of the AST of each model. In the base model, the top-level object is COIL-Program. Every element of the COIL is ultimately a component of the COIL-Program. In the DOM everything is ultimately a component of the DomainTheory class. To merge the two models, it was determined that these two objects must be considered semantically similar (according to step two of the model merging process). Therefore, the COIL-Program will be retained and used to represent domain theories, requirements specifications, as well as designs in the new model. That is, if the AST is a domain theory, an instance of an AWSOME Program will represent the root of the domain theory AST. Likewise, if the AST is a design, an instance of AWSOME Program[1] will represent the root of a design AST. A standard naming scheme for AWSOME objects was adopted. It

[1]The function of the root node was later enriched, generalized, and renamed as Package (i.e. WsPackage)

prefaces each class in the AWSOME model with the indicator, Ws, for "Wide Spectrum." Figure 24 shows the conceptual process followed to combine the top-level objects of the COIL and DOM.

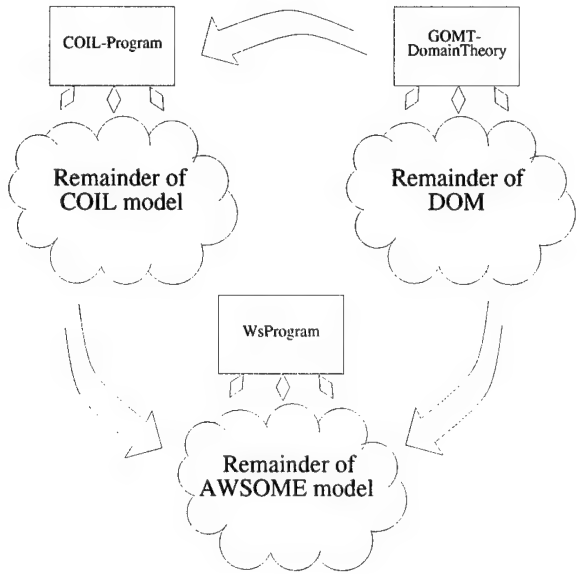

Figure 24. The merger of the AST root of the DOM and COIL

In the top-down approach followed, the next step was to examine the component classes of the DOM and determine their position in the COIL, according to our methodology. The components of a DomainTheory (predefined types, global user-defined types, classes, and global constants) were then integrated into the COIL model.

4.1.1 Type System. The COIL already contained an extensive type system that allows users to create any types necessary. However, the COIL did not separate the semantic concepts of predefined and user defined types. To include this DOM concept, predefined type was added to the COIL using a combination of steps 2b and 3 of the methodology.

During the development of the COIL, Graham determined that classes "behaved" very much like types; therefore, he included classes in the type system. This concept was

retained in the creation of the wide-spectrum model. More detail of the AWSOME class structure will be discussed later. The AWSOME type system is shown in Figure 25.

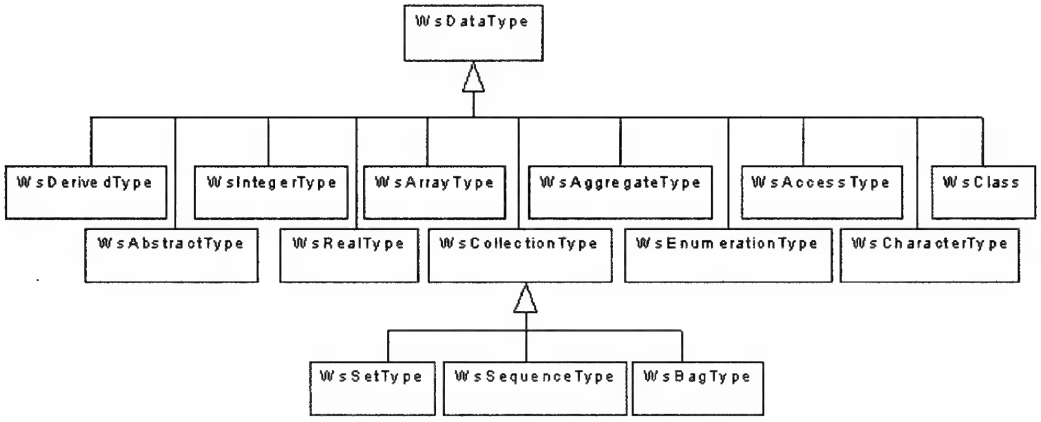

Figure 25. The AWSOME type system

4.1.1.1 Example AST. Using the AWSOME type system, we can show how an arbitrary type can be instantiated. This will demonstrate the instantiation of an integer. Before an AST can be shown, the model for an integer type must be discussed. The integer type will have a name. It also has upper and lower bounds. These bounds can be set to model any desired behavior. For example, if a 64-bit implementation of a signed integer is to be modeled, the lower bound would be set to $-9.223372037 \times 10^{18}$ and the upper bound to $(9.223372037 \times 10^{18}) - 1$. In this model we use Literals to store the actual values of the upper and lower bounds of the integer. WsLiteral is an expression, WsExpression, and is discussed later in Section 4.1.4. The model of an AWSOME integer is shown Figure 26.

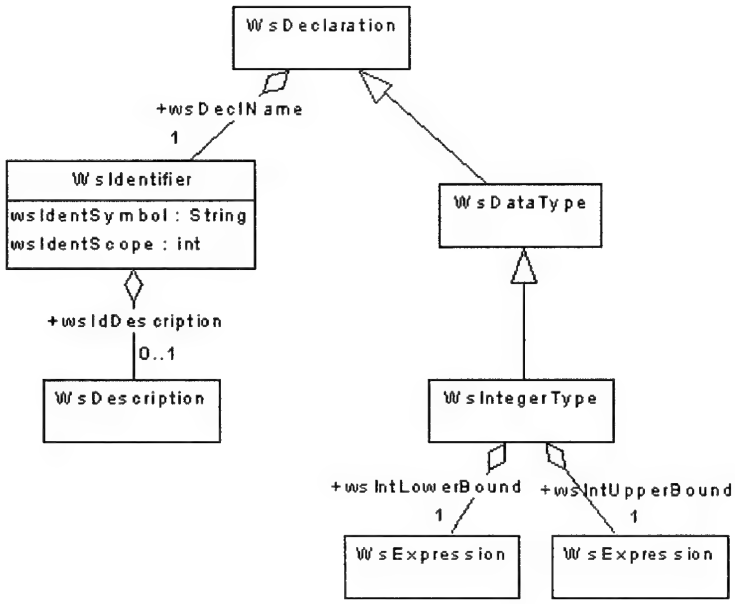

Figure 26. An AWSOME model for integer types

C integer types `int`, `long`, and `short` instantiated in the AWSOME have ASTs as described in Figure 27. This AST for a C integer types can adopt the COIL surface syntax as its AWSOME syntactical representation:

```
type short is range -32768 .. 32767;

type long is range -2147483648 .. 2147483647;

type int is range -2147483648 .. 2147483647;
```

4.1.2 Constants. The next component of a Domain Theory of the external model is global constant. The COIL already provided a construct for constant. Using step 2a of the methodology, we will allow the COIL construct for a constant to take on the semantics the DOM global constant.

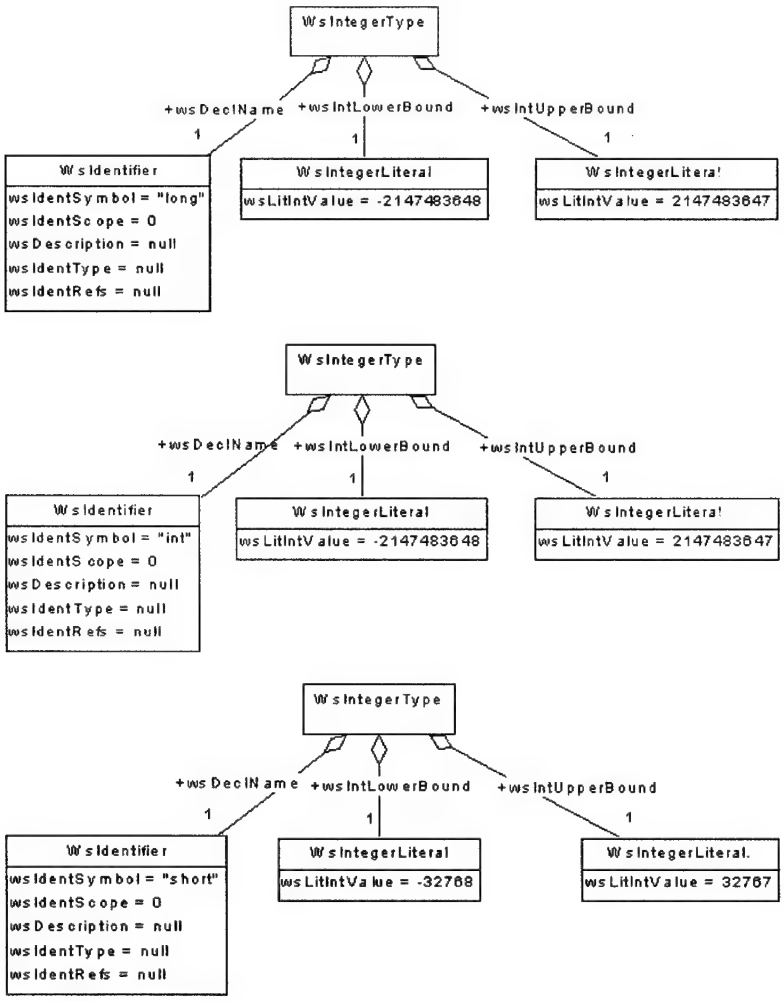

Figure 27. AWSOME ASTs for C integer types

So far, we incorporated three of the four DOM top–level components into the COIL with minor adjustments or using a common structure to represent similar semantic concepts. The remaining top–level DOM concept was the class model.

4.1.3 Classes. As stated earlier, the COIL represents classes within its type system. The COIL models a class as an aggregation of attributes, methods, and a super class. The DOM was not designed to represent programs, but object-oriented formal specifications. A DOM class contains some common semantic concepts such as attributes.

We note the equivalent semantics of attributes in both models and apply step 2a of the methodology. Similarly, we allowed DOM operations to be semantically equivalent to COIL-methods.

Next, DOM types are considered. The COIL has only global types. Since the objective is to represent a *core* language for software synthesis, we elected to eliminate local types, allowing them to be represented as global types in the AWSOME[2].

A chief representation concept of the DOM that had to be integrated into the COIL-class to form the AWSOME class model was the predicate logic used to describe formal domain theories and requirements specifications.

4.1.4 Predicates. The COIL did not have a representation of predicates. The DOM used a predicate AST based on the structure of the formal language, Z [26]. Before predicates could be added to the COIL, an appropriate representation for the syntax and operations of predicates had to be added to the COIL. Using our methodology, we recognized the COIL contained math operators in common with some DOM predicate operations, e.g. $=$, $\leq$, $\geq$, etc. These DOM predicate operators were modeled as math operators in COIL-expression. Step 2a of the methodology allowed these common operators to be declared semantically equivalent to those of the DOM. All that remained was to extend expression with the remaining DOM predicate operations. For demonstration purposes we added a basic set of operations. These include logic operators such as $\forall$, $\exists$, and $\Rightarrow$. Also included were set operations such as $\subset$, $\subseteq$, and $\in$. Finally, set former capabilities

[2]If local types are desired for a particular target language, they can be created dynamically during code generation

were added using step 3 of the methodology. This provides enough capability to formally express most predicates. AWSOME Expressions are shown in Figure 28.

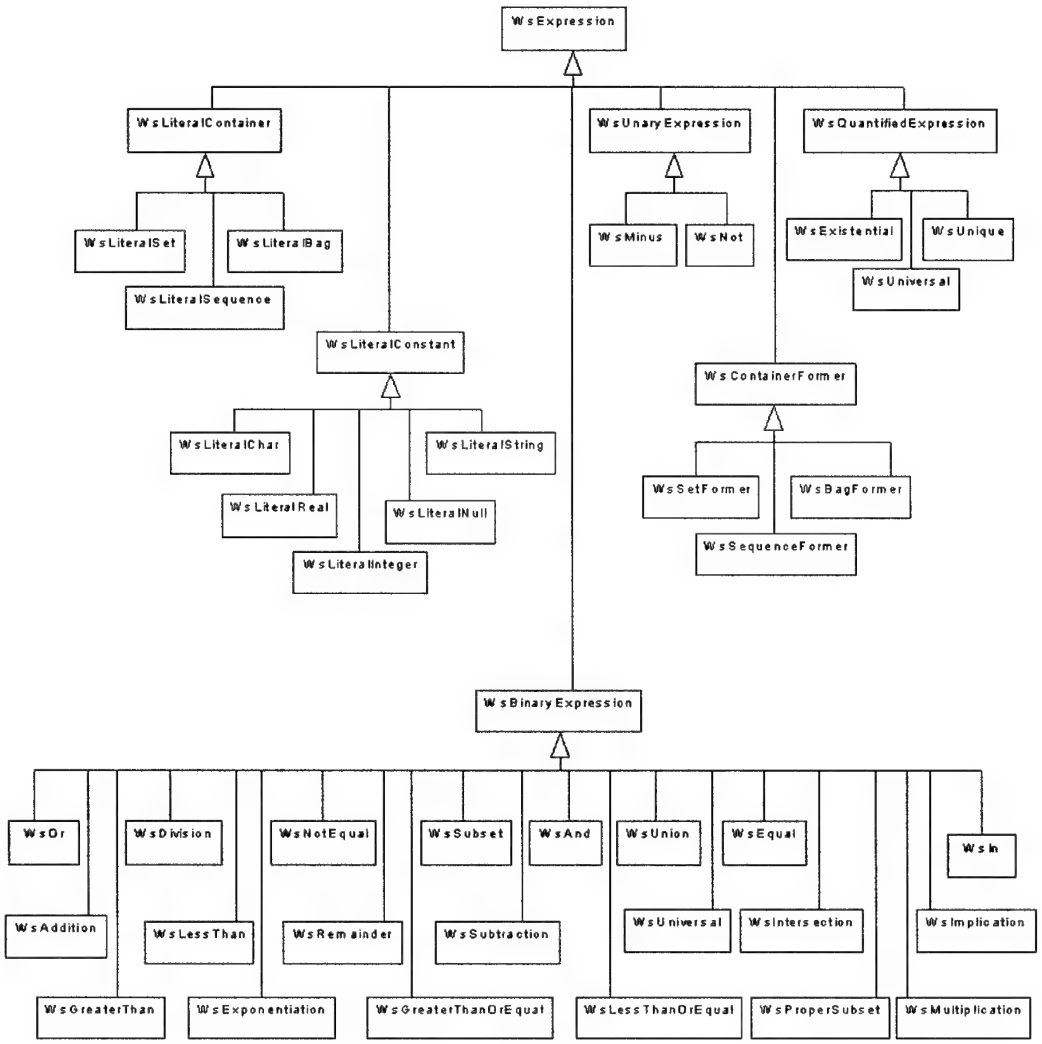

Figure 28. AWSOME Expressions

Once predicates have been added to the AWSOME, their usage, along with other DOM concepts must merge into the COIL. This meant adding the capability to represent pre– and post–conditions in COIL–methods for the wide-spectrum model. Additionally,

DOM invariant constraints were added to the AWSOME class. The bold lines in Figure 29 indicate how predicates were "pasted" into the COIL–class to create the AWSOME representation of a class.

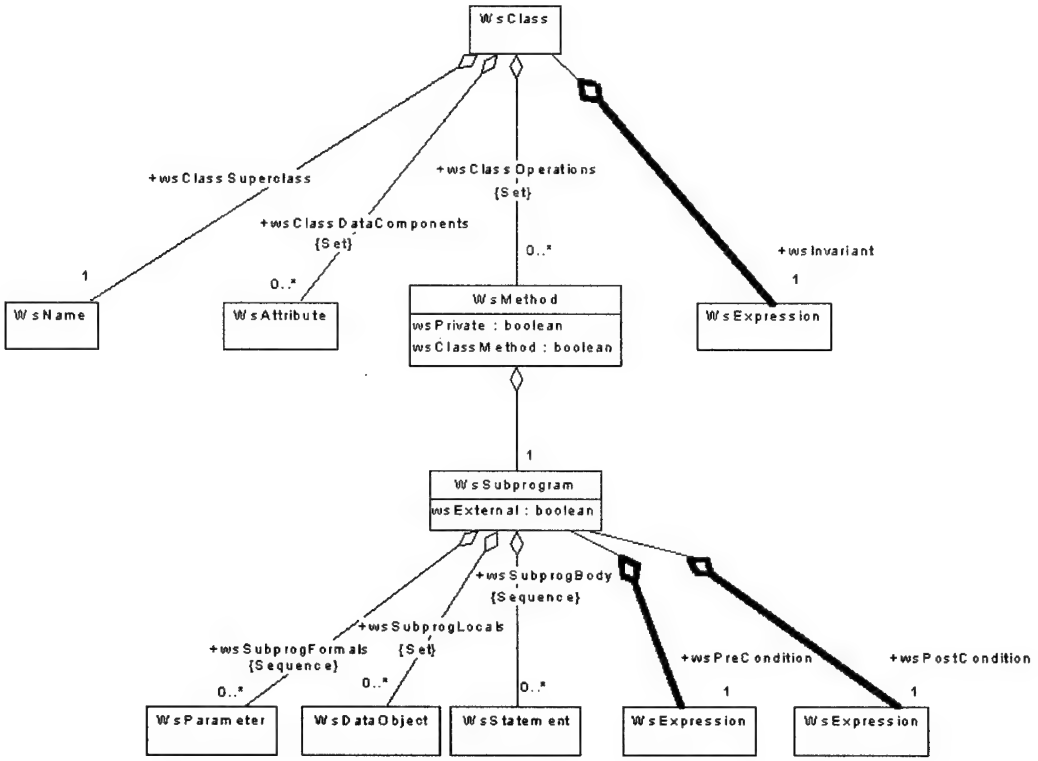

Figure 29. Adding DOM predicates to COIL classes creating AWSOME classes

4.2 Dynamic Model and Associations

Two final semantic concepts of the DOM were merged into the COIL. These were the Dynamic Model and Associations. In the first draft of the AWSOME both concepts were added according to step 3 of the methodology. As a way of collecting the semantics of the Dynamic Model, one component class, WsDynamicModel, was added to WsClass. Similarly, WsAssociation was added at the top level of the model, that is it has no super

class other than WsObject in the AWSOME model. Finally, the EventMap, a component of a class in the DOM, was made a component of class in the AWSOME model.

During this research, ongoing parallel research was investigating improved models and new transformations for both Associations and the Dynamic Model [11, 33]. The reader is encouraged to review the other research for the final implementation of these concepts in the AWSOME model.

4.3 Summary

The creation of the AWSOME model involved a careful consideration and migration of the semantic concepts of the DOM. Each DOM component was mapped to or added to the COIL as necessary. The resulting model was renamed the AWSOME model. Key components of the AWSOME model are described in Appendix B.

The AWSOME model became the information model for the software synthesis repository. Once this model was in place, the research could proceed to the development of a prototype repository and the application of the relationship methodology proposed in the previous chapter.

V. Implementation

This chapter discusses repository prototypes and proofs–of–concept implemented as part of this research. The repository system developed will be used primarily in a software synthesis environment. Consequently, the repository was developed with a focus on client software synthesis tools. However, we believe the concepts of this research can benefit any application of repository technology.

An overwhelming number of features are required for the implementation of a full-scale repository. The time constraints for this research allowed the realization of a limited number of these features. The features implemented focus the effort toward the research goals of promoting the understanding and reuse of software synthesis artifacts, and showing how repository relationships can be used to record design history information automatically.

This chapter discusses repository design considerations and repository-specific extensions added to the AWSOME model to make it "repository aware." Additionally, we discuss the implementation of portions of the repository engine—the focal point of which is the repository meta-model. Specifically, we focus on the implementation of relationships used to organize the repository. After that, the chapter shows how repository relationships in an OODBMS context can promote understandability and reuse. Finally, we show how these relationships can be used to represent design histories.

5.1 Design Considerations

For this implementation, limiting factors included available development tools as well as the software synthesis tools developed during previous AFIT research. The development tools selected and implementation approach taken are influenced, in part, by this previous research. The architecture of our system is described in the next few sections.

5.1.1 System Architecture. As with any implementation of a software system, the repository system was constructed within a specific architectural framework. Though elements of Chapters II and III foreshadow the architecture by showing repository architecture in the general case, this section discusses the specifics of our implementation.

5.1.1.1 Client/Server. In a very early prototype of the repository, an information model based on a simplified GOM was implemented in Java. In this prototype, a socket-based, client-server model was used. That implementation was attractive for the final versions because it provided flexibility to the repository for existing, Unix-based client tools and future Windows-based client tools. Expanding upon this approach will allow client tools to share the repository across heterogeneous platforms.

The early prototype was extended into the current version by adding several desirable features, such as thread-based handlers for client requests and the ability to provide limited repository administration. The socket-based nature of the interface made it necessary for the client and server tools to agree on specific communication protocols. Since these protocols are not the focus of this research, they are not extensively discussed. However, these protocols were improved significantly throughout the development of the repository.

Information about these protocols have been hidden from the client tool developer by the Message class discussed in Section 5.2.2.

5.1.1.2 OODBMS. When this research began, the object-oriented database management system currently licensed by AFIT, Object Store v5.1, was available only on the institution's Unix platforms. The OODBMS is heavily based on C++, but provides a Java interface, as well. In later phases of the research, AFIT acquired the Windows edition of Object Store v6.0. This version came with a number of graphical design tools and an improved Java interface. The graphical tools were used to generate portions of the repository meta-model.

It should be noted that the normal process for developing Object Store friendly client applications is to install the Object Store client on each client machine that will use the Object Store database server. AFIT did not license the ObjectStore client for version 5.1, consequently this research developed a methodology to overcome this deficiency. The work- around will be discussed in Section 5.2.2.

5.1.1.3 Implementation language considerations. As discussed in earlier chapters, the previously existing tools and models were implemented in a proprietary programming language. This language, called CQML, is part of Reasoning5, a Code-base Management System [41]. *CQML*, formerly *Refine*[1], is implemented over an Allegro Lisp [21] runtime system. One secondary goal of this research was to divorce the current software synthesis system from this proprietary language and use a more common language such as Java or C++.

[1]We refer to CQML as Refine for the remainder of this document

Java has been the language of choice for much of the latest KBSE tool development at AFIT. Since these tools would be using the AWSOME model as their object model, it made sense to implement the AWSOME model in Java. These two considerations made the Object Store Java Interface (OSJI) the natural choice for implementing the repository information model. The Java code developed originally for the AWSOME model could be ported to database code with only minor modifications necessary to comply with Object Store persistent class capabilities. Powerful Java classes for socket-based communication also simplified the programming of the socket-based interface discussed earlier. Finally, it allowed us to leverage the work done in the earlier prototype.

5.1.2 External interface for existing tools. One of the early tasks undertaken in this research was to provide an interface for the existing tools. This proved difficult, but possible. The complication of Refine's lightly documented Lisp interface, along with the specifics of interfacing an external language to Allegro Lisp proved challenging until we discovered the respective correct syntaxes. With the correctly formatted Refine programs, and proper compiling and linking of the external language, we demonstrated how the Refine interface with Lisp could be used to call the Lisp interface to external languages, such as C.

Specifically, we used these interfaces to pass parameters from Refine to C, and receive results from the C programs. Using this technique, we can interface with many common programming languages either directly or via an external interface through a C program.

This demonstration convinced us that current tools could be modified to interface with an external repository[2]. A detailed description of the process used is shown in Appendix A.

5.2 Repository engine implementation

5.2.1 Tool Interface. As stated in Section 5.1.1.1, the initial repository interface developed was a cross-platform, client-server implementation. Since cross-platform communication can be costly, we did not want to restrict more tightly coupled repository clients—that is, tools that run against the repository directly. We desired to develop a limited cross-platform, socket-based interface, as well a more robust interface for tools that might be designed to run "directly on" the repository information model. This might be considered to analogous to the "thin net" and "thick net" approach used in the development of JDBC[3], the Java interface to SQL databases. The second, direct interface was scheduled later in the development; as such, only a very few, limited directly called methods were implemented. The methods implemented serve only as a proof-of-concept of the design history methodology.

The socket-based tool interface developed was very simple. The Repository Engine was implemented as a server. The server (a ServerSocket in Java) waits for client connections, services the incoming client requests, and sends the appropriate response.

A very simplistic repository administrator client was also developed. The administrator client requests connection with the repository server, sends a request, and waits for a response. Upon receiving a response the administrator client will remain connected to

[2]However, part way through this research the research group decided to re-implement most of the previous tools in Java making this interface unnecessary.

[3]JDBC™ is not an acronym, but is often referred to as "Java Database Connectivity."

allow for other administrator requests. This implementation allows the administrator to block any "regular" tool clients during ongoing repository administration. These regular clients will be expected to disconnect after each response from the repository server.

5.2.2 Message Class. The repository interface is implemented as a single Java class, Message. The Message class is an aggregate class. It contains an attribute, Command, in which the client request is sent. It contains the Vector of parameters that optionally accompany the client command. Also, Message contains a component class, Object, in which the repository response will be sent back to the client. The client will be required to type-cast the response to the expected type. Finally, it contains a string attribute, sender, which provides the capability to distinguish the administrator client from other client tools.

The Message class is used by three main methods: listAST, putAST, and getAST. These methods currently allow the user to list the artifacts in the repository by name, retrieve an artifact from the repository, or place an artifact in the repository. All three methods currently operate based on name alone. Obviously, a fully-implemented version would take advantage of the many query capabilities of the DBMS.

```
public class Message implements Serializable{
     private String sender = "user";
     private String command = "listAST";
     private Vector parms = null;
     private Object msgContent;
}
```

The put method causes transient AWSOME instances to be stored as persistent instances in the Object Store database. Object Store normally requires these client-based instances to originate from client systems on which the Object Store client is installed. As

stated earlier, AFIT did not license this client software initially so an alternative method for making this data persistent was developed. The repository implements a persistent copy operation. The persistent copy "visits" every node of the transient AST passed from the client and makes a persistent duplicate for storage in the DBMS. Likewise, a client request for a transient object from the database walks the persistent abstract syntax tree and instantiates a transient copy to be passed to the client.

5.3 Meta-Model Implementation

The repository meta-model is a part of the repository engine. Since so much of this research focused on developing a meta-model that promotes reuse, especially in the AFIT software synthesis environment, this section is dedicated to its implementation.

5.3.1 Repository Artifact. To promote reuse, a methodology for uniquely identifying artifacts was develped. To do this, an artifact object was created. WsArtifact was initially given the following structure.

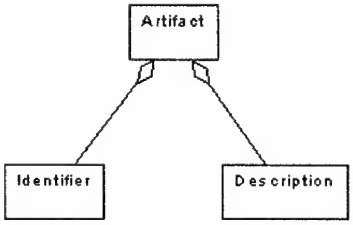

Figure 30. Initial model of a repository artifact

The WsArtifact includes a description object, WsDescription, and an identifier object, WsIdentifer. As the research progressed, AFIT KBSE researchers realized the value of description. We determined the potential for understandability of an artifact might be

increased by providing a way for *any* identifier to contain additional descriptive information. This was achieved by making a slight adjustment to the description object (shown in Figure 31).

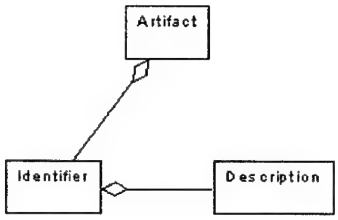

Figure 31. Model of a repository artifact

5.3.2 Repository Composition. Previous chapters state that a repository is a collection of artifacts and the relationships between those artifacts. At the top level of our repository engine, we describe this concept in the object model shown in Figure 32.

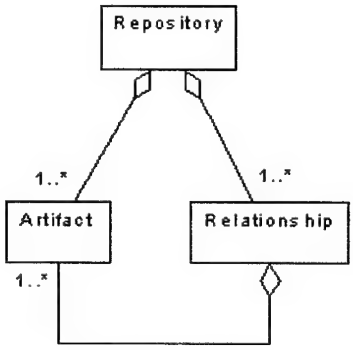

Figure 32. Model of a repository engine

5.3.3 Repository Relationships. The model for repository relationships was presented in Section 3.4.2. Here we discuss some of the points of implementation.

The user must have the capability to add new relationship instances to the repository, as well as add and remove artifacts from these relationships. Consistent with any

container class, we define such methods as addArtifact, removeArtifact, addRelationship, and removeRelationship. These methods can be inherited by any new subclass of relationship that may be defined. The methods are shown in Figure 33.

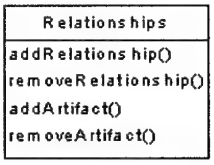

Figure 33. Four main methods of Relationship

Since each identifier has a description, every instance of an identifier in the AWSOME model becomes meta-data available to the repository. Information retrieval tools can use this meta-data to assist a repository user in locating and understanding a reusable artifact.

5.4 Making the Information Model "Repository Aware"

Some previous AFIT KBSE models uniquely identified ASTs with a name string stored in the root node of the tree. The COIL did not provide an internal representation of a name for each AST since a unique name could be give to the text file containing the COIL surface syntax. The AWSOME model originally inherited this characteristic of its COIL ancestor, making it what this research terms *repository unaware* .

The AWSOME model is made *repository aware* by adding just enough meta-data to provide basic information required by the repository meta-model. This research defines three possible ways to populate this repository meta-data: 1) fully define the repository meta-model in the information model, 2) automatically extract tool data for the repository meta-model, or 3) provide a hybrid of the first two. First, for the repository meta-data to be defined in the tool's information model, the repository meta-model could be "merged"

into the tool information model. This option requires tools to provide all the meta-data as part of their inherent model. The repository will then copy the data from the information model instances to the repository meta-model instances. Second, the repository engine could infer (or request) all necessary meta-data to build instances of the repository meta-model. No knowledge of the repository is included in the tool's information model. The data is intelligently extracted, or provided by external user input. Third, a hybrid of the two could be possible providing just the core information the repository needs to store unique AST instances. This research termed the third method repository aware.

To make a repository capable of storing many, possibly related AWSOME ASTs, each AST must be uniquely identifiable. Additionally, to promote reuse, other descriptive information should be available. The methodology for retaining this development information involved improving the artifact object of the repository meta-model, WsArtifact, with descriptive information. This included a WsIdentifier and WsDescription. This additional meta-data makes the AWSOME model repository aware.

These include the artifact, description, and repository objects discussed in the previous paragraphs. The structure of these objects, as represented in the repository engine, is discussed in Section 5.3. The rationale for choosing these objects to make the AWSOME repository aware is discussed in the following sections.

5.4.1 Repository. A repository object is added to the AWSOME model to allow tools to have the option of manipulating collections of artifacts. The addition of a WsRepository class to the AWSOME model will now make the repository object the new

top-level object—the new root. That is, the AWSOME model can represent a collection of artifacts that are, in turn, collections of declarations.

This research does not refer to the new root as the root of an AST, however—each artifact instance could be an AWSOME AST (or some other artifact). The model should be able to maintain the semantic concept of operating on one or more AWSOME ASTs. Additionally, a tool may be designed to perform activities on artifacts belonging to separate information models (as previous AFIT KBSE tools did). We have advocated the benefits of using a single information model, but don't want to restrict client tool implementations by making a single information model mandatory. The repository object, WsRepository, allows a single root to represent a collection of ASTs.

5.4.2 Artifact. The artifact object provides two essential concepts for repository awareness, identifier and description. Description is discussed individually in Section 5.4.3. Identifiers are already an inherent part of the AWSOME model. They provide a name and other useful meta-data about AWSOME declarations. The inclusion of an artifact in the AWSOME allows the AST root to inherit an identifier from the artifact.

With this addition, AWSOME-based tools can now use more than one artifact represented in a shared information model (consider the Elicitor-Harvestor scenario in which more than one domain theory is combined to generate a requirements specification). Additionally, other artifacts, such as documentation, designs, or architectures have a "built-in" location in the AWSOME model.

5.4.3 Descriptions. One chief aspect of understanding and reusing artifacts is thorough documentation of types, attributes, classes, etc. The intent of the description class is to contain additional information about any given identifier. Just like the name of an identifier, the description of that identifier can serve to aid in selection and reuse of an artifact.

It is this utility that encouraged us to add this attribute to the AWSOME model. The description will aid Elicitor-Harvestor users who work with as little as a single AST, as well as repository users evaluating a number of artifacts for potential reuse.

The data within description and identifier instances will provide the data that information retrieval tools can use to infer meta-data about repository artifacts. Data from these objects can be extracted to build structures required by the variety of representation methods used in a repository engine. More about how this is done is discussed in the next section.

5.5 Representation Methods

The data from the repository relationships developed in this research can be used to populate the meta-data necessary for many of the representation methods discussed in Chapter II. Additionally, representation methods can be modeled directly with repository relationships.

For Information Retrieval (IR) tools to make use of faceted classification, they may simply extract their meta-data from information in instances of the identifier object. This

object will provide the name of each identifier, its type (when applicable), description data, etc. A domain analyst should have everything necessary to classify domains.

Not only will this meta-data be useful for IR, but AWSOME tools may use data modeled directly in repository relationships to represent domain classifications. A domain is a relationship object in which multiple artifacts are considered components of the domain instance. In fact, the similarity relationship (Section 3.4.2.1) outlined earlier is a generalization of a domain. (In the old AFITTool, classification into domains was accomplished by storing all the files containing information about a single domain in a common directory.) Because of our OODBMS implementation in Java, we can allow a single Artifact instance to participate in multiple Domains. In fact, multiple domains can participate in other domains, as well (See Figure 34).

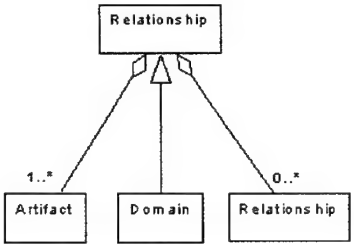

Figure 34. Relationship for domain classification

Hartrum described domain theories for simulations of a traffic light [28] and queueing system [27]. Both simulations require integration with the domain model of a simulation system [27]. The simulation system domain theory models a general-purpose discrete event simulation system. Under the previous AFITTool scenario, the traffic light model is represented by several files in the traffic directory. The queue model could exist as several files in a second directory. Finally, the simulation system model exists as a file in yet a

third Unix directory. It is clear that both the traffic light and queue models require the simulation domain theory, but this can only be inferred by the engineer examining those models in detail.

Using the relationships developed in this research we can show the artifacts related to each domain theory, as well as the relationships between domain theories. Consider the instance diagram in Figure 35. It shows the instances of the various artifacts, the domains in which they participate, and the relationships between those domains. This kind of meta-data could be very valuable to the tool developer desiring to present a complete picture of the interrelationships between domain theories. It is important to note that each instance need exist only once, but may have many references to it.

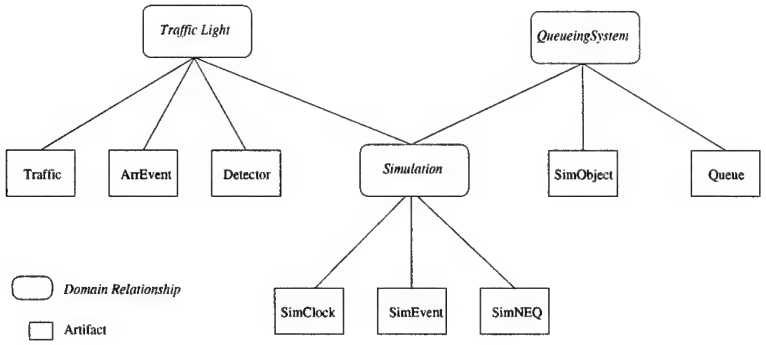

Figure 35. Domain classification

5.6 Implementing Design Histories

One of the most complicated relationships to implement was the nested one that is the nature of a design history. We expected a design history meta-tree to be costly in terms of space. Future implementations can attempt to store only changes to the object models. As a small, initial demonstration, we created a design history instance "on paper." This instance is shown in Figure 36. This transformation example, derived from Smith [45],

is an example of a design history meta-tree resulting from the application of algorithmic transformations[4]. In this example, a software engineer is nearing the completion of a design. The final activity is to apply a sorting algorithm. The software engineer first chooses the global search transformation—a non-viable solution. The engineer then successfully applies divide-and-conquer techniques. The first application results in an inefficient insertion sort. The "n/2 split" results in a more efficient merge sort. The software engineer selects the merge sort AST as the completed design.

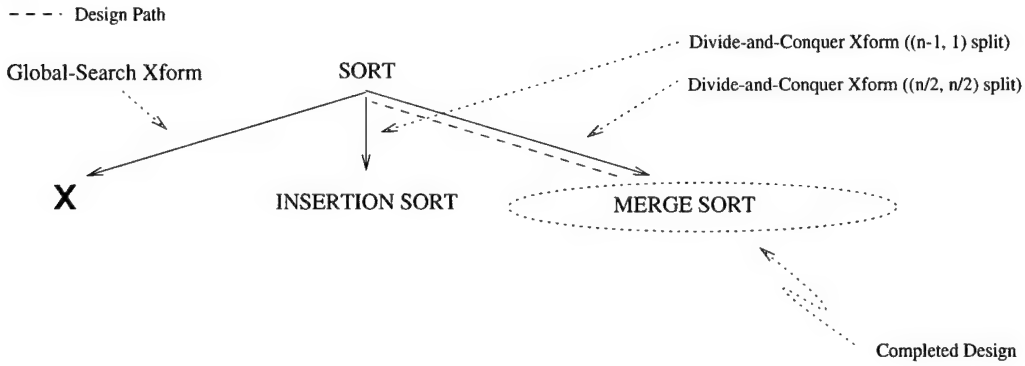

Figure 36. Initial concept of a design history instance using an algorithm transformation example.

Using this basic idea, the next step was to determine the kind of meta-data about a design history that should be retained. For example, the transformation that led to the current intermediate (or final) step and the design rationale used to select that transformation are key to future understanding of a design. A detailed object diagram of the design history relationship is shown in Figure 37.

The design history is implemented as a type of repository relationship. As such, it will inherit component artifacts. The history relationship will be constrained to a single artifact,

[4]These transformations have not yet been implemented by AFIT KBSE researchers.

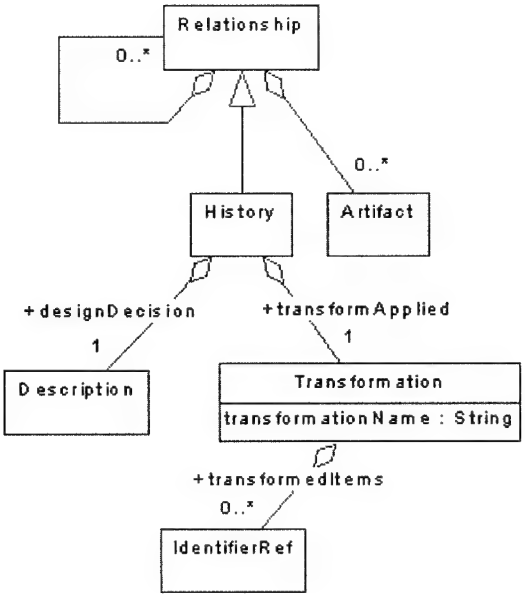

Figure 37. A design history node in the design meta-tree

however. It will also inherit component relationships. The relationship will represent the path to all subsequent history nodes in the design meta-tree.

The meta-data to be retained for each history node must be considered next. The chief component of the history relationship is the Transformation class. This class will contain the transformation that resulted in the creation of this history node instance—that is, the transformation *previously* applied that resulted in the current version of the AWSOME AST. Each history also has a component description in which the engineer (or transformation system) can document the design rationale for applying the chosen transformation. Finally, a transformation object contains a component reference. This reference *may* be used to "point" to AST components selectively chosen for transformation (not every eligible AST component must be transformed—an engineer may choose to apply transforms to a subset of AST components, rather than the entire AST).

Relationship methods were used to create and link nodes of the design meta-tree. The methods were extended with methods to populate and query the designDecision and transformApplied component classes. The design meta-tree resulting from the algorithm example (Figure 36) is shown in Figure 38. The implementation of example design history within the transformation system can be the subject of future endeavors.

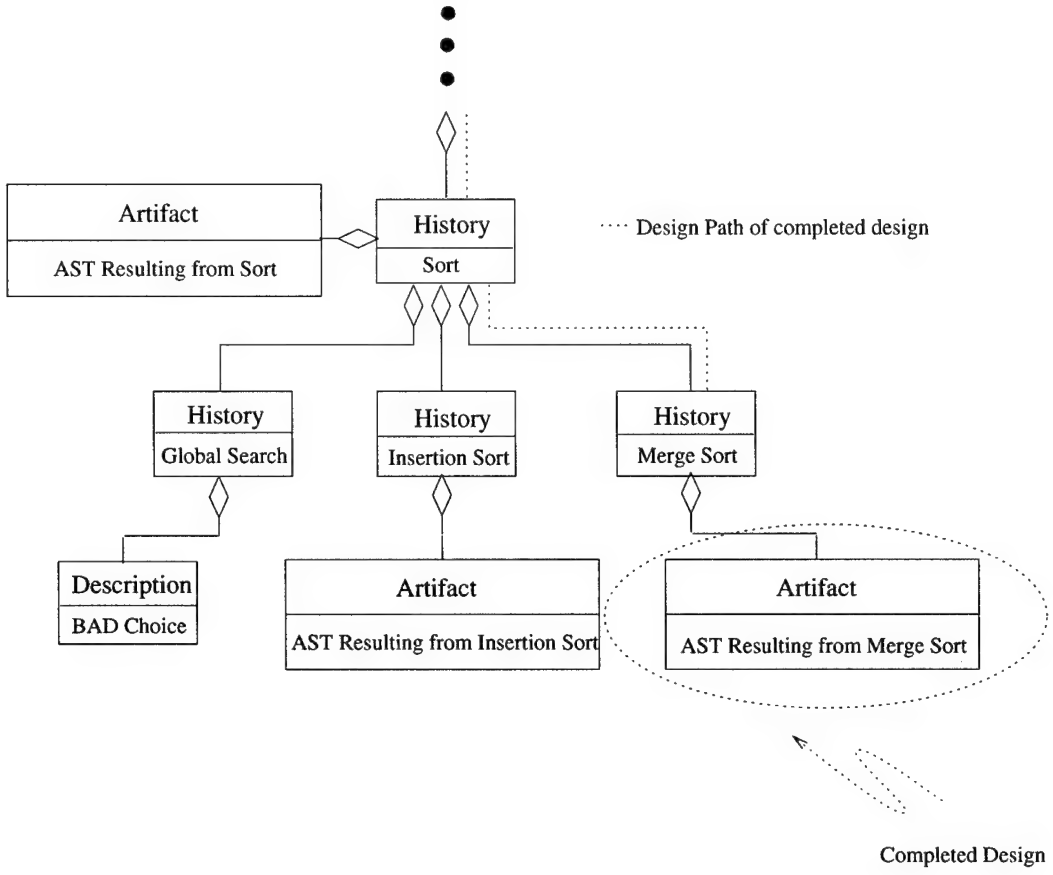

Figure 38. Design history instance represented in a design meta-tree

5.7 Conclusion

In this chapter a number of beneficial concepts for improving the software synthesis process were shown. Through a repository-based process, the addition of new tools is made

simpler. By using repository technology the representation of relationships between data can benefit software synthesis tool users—and tool developers.

For tools to share data easily, they should use a common object model. In this case, the AWSOME model was developed by combining (then extending) existing, specialized software synthesis models. The methodology for combining these models was derived from an examination of the combination of other software synthesis models.

Finally, repository relationships can be used for a variety of purposes, including representing design histories. We represent design histories as a tree of related artifacts—themselves trees— and call it a design meta-tree. The root of the design meta-tree is the requirements specification and one or more leaf nodes is a completed design.

The representation of these concepts in a repository should achieve the goals of improved reusability, and begin to solve the software synthesis problem for representing design histories.

VI. Conclusions and Recommendations

The focus of AFIT KBSE research historically has been transformation theory, and not on managing the large amount of data that transformation systems create and use. The software synthesis process will reuse previously developed domain models. Since system users will have to locate and understand the applicability of the domain model, this is not simply a database problem, but an information retrieval problem. Repository technology has been applied successfully in environments where reusability is of primary importance.

This research implements components of a repository, then discusses and demonstrates the benefit this technology can have in the AFIT software synthesis environment. Section 6.1 details the results of this research.

We have only begun to introduce this technology in this setting—much more repository work can be done. Future research can be applied specifically to the software synthesis problem, or can be applied across the spectrum of Air Force applications dealing with the management of meta-data. Possible future related research is discussed in Section 6.2.

6.1 Results

6.1.1 Repository-based software synthesis. First, it was necessary to frame the system in a repository-based concept. Our research modified the software synthesis process making it repository-centric. Among the benefits are centrally managed and easily sharable data. Additional repository meta-data can describe relationships between a variety of artifacts, all of which promote a key goal of software synthesis—reusability.

An environment that allows data to be shared works best when the data is represented in a common object model. The repository-based process proposed requires a common object model for software synthesis. This is a new way of thinking—previous AFIT KBSE research used one model for domain analysis and requirements specifications and another to represent designs. The previous software synthesis process was designed around these models.

6.1.2 A repository information model for software synthesis. This research proposes a "wide-spectrum model" for software synthesis. This model, called the AWSOME model, was based on a concurrently-developed common object model, the COIL, built by Graham [24]. The COIL combined two previous AFIT software synthesis models. By analyzing the COIL, and comparing it to its predecessors, a methodology for combining two object models was inferred[1].

The methodology we developed combines object models by "adding" an external model to a base model. This process captures the semantics of the external model in the base model, but not necessarily the structure (syntax) of the external model. Previous AFIT research [44] was concerned with capturing precise syntax of, and recreating instances of external object models. This precision was not necessary for this research. Already, this methodology has been considered for adding additional semantics to the AWSOME model [50].

Six AFIT researchers were involved in developing the AWSOME model as it is today, but this research developed the original, base AWSOME model using the methodology pro-

[1]This was not necessarily the method used by Graham, but it worked for purposes of this research

posed. This process successfully captured the previously-existing semantics of the external language. For example, expressions in the external model were present in the base model, also as expressions. The external language represented set-theoretic expressions, so the semantics of set theory were added to the base model—in the expression portion of the COIL. Once the combination of models was complete, the resulting model was named the AFIT Wide Spectrum Modeling Environment. Like its parents, instances of the AWSOME model exist (at least initially) as abstract syntax trees.

Once a common "wide spectrum" model was in place, we could build a repository that uses it as the repository information model. In repository terminology, the object model is known as the information model. This research built a persistent object model based on the AWSOME. AWSOME instances were stored, retrieved, modified, and updated in the repository.

6.1.3 Repository engine for software synthesis tools. The repository engine provided import and export capability to a transient version of the AWSOME. This allowed AWSOME-based tools to have the option of using the repository information model directly, or to import/export from existing tools based on the transient version of the AWSOME.

6.1.4 Relationships within a repository. The research added meta-data to the repository engine to represent relationships and promote understanding of the reusable artifacts in the repository. Besides artifact identification and description, the research generalized a previously existing repository relationship model developed for [13]. The effort then developed a number of specific relationships based on the general relationship

representation. We demonstrated a repository organization designed to promote understandability and reuse.

6.1.5 Software synthesis design histories. Finally, the research showed how one specialization of a relationship, *history*, could be used to represent the concept of a design history. The history relationship relates one artifact to another artifact from which it was derived. The history relationship contains information related to a software transformation system. This information includes the transform applied, the design rationale for the choice of that transform, and references to the AWSOME identifier of each AST component transformed.

We showed that in transformation-based software synthesis systems design information will exist as a design meta-tree of intermediate designs with the root as the requirements specification, and at least one leaf node as a completed design. Intermediate nodes will represent incomplete or perhaps undesired designs. Once a design is complete, design tools may offer the option of saving only the sequence(s) of transformations leading to a completed design (or multiple design alternatives), or selectively saving additional data about the exploration of the design space either as future design alternatives, or as a warning to others to stay away from a certain design path. We suggest that the user be presented this design meta-tree and should have the option to prune this "tree of trees" as he or she sees fit (though the sequence of transforms from the root to the completed design are saved automatically as the design history). This is unlike a previous AFIT transformation tool, Elicitor-Harvestor, where the transformation history is a simple sequence of

all transformations explored from the initial state to the final state (similar to the history command on Unix).

6.2 Future Research

During this research effort, a number of new potential research area came to light, or were scoped out of this project due to time constraints. These areas are recommended for future study. The potential future research areas identified are listed in the following paragraphs.

Repository Engine: One of the most obvious areas for extension is the implementation of the repository engine. Many features identified in earlier chapters were simply not explored in this implementation. For example, the management of workflow models was not explored. Also, additional repository relationships, as described in Chapter III could be added to aid in version and configuration management.

OODBMS Issues: During this research much ongoing work on object versioning was encountered. Research into the improved methods for identifying and efficiently storing versions of object instances is an entire field of research unto itself. Since every new instance of an object should not necessarily be considered a new version, work is underway to determine how to best version object instances.

In OODBMSs, object instances can be stored persistently in special collection classes called extents. Extents can exist outside the "regular" object model. Extents can be beneficial since all instances of a particular class can be stored internally to the database in sets, sequences, or other easily-searchable container classes—regardless of where they

exist in the hierarchy of the object model. Extents could be used to more easily search such meta-data as AWSOME WsIdentifier or WsDescription extents to aid users in locating and understanding reusable objects. Additionally, the data in extents could serve as part of, or be used to populate the repository meta-model proposed in this research.

Other databases: Often the peculiarities of making objects persistent in an object-oriented database were troublesome. Future researchers could explore the benefits of representing information models in a relational database. For example, a relational schema representing the AWSOME model could be developed. Once both models were in place, objective comparisons of repository performance and ease of implementation could be explored. Toward the end of this research, AFIT acquired Oracle 8i. It is reported to include many powerful, automated information retrieval features. The features automate the creation, indexing, and population of meta-data. Additionally, it has an object-oriented front-end which might simplify the transition to this database and allow the migration of our repository engine.

Software synthesis: One of the benefits of using an object-oriented information model is having the repository provide the object model to be used by client tools. Currently, all AWSOME-based tools use their own implementation of the AWSOME model, and can import and export from the AWSOME information model in the repository. Researchers should explore the implementation of software synthesis tools that use the information model directly—after all, one purpose of a repository is to provide a common implementation of the information model for all tools to share. For example, transformations would be applied directly to the persistent ASTs in the repository. Tools could also take advantage of database queries, and the repository meta-model to aid in enhancing a

user's ability to select and reuse existing software synthesis artifacts in new applications. For instance, an elicitor-harvestor tool might focus on the development of many new and powerful features if its underlying information-retrieval functionality could use repository meta-data, and database search tools.

Additionally, this research solved only part of the design history problem—the representation of design histories. More research is required to determine how best to use this representation of a design history to "replay" designs. The theoretical goal of design histories as applied to KBSE is to enable modifications to requirements to transform automatically to a new design version by simply replaying the original design.

Other uses for repositories: This research should be extended to other applications of interest to the Air Force. Most promising is a repository of simulation models. AFIT is currently researching ways to represent a variety of different simulation models in a single global model from which tool-specific schema models can be generated. A repository would be useful to store and manage simulation scenarios, the simulation models, or the transformations between the global and tool-specific object models. Similar to transformations between AST instances in software synthesis, the transformation between a global schema and a tool schema can be represented as repository relationships.

Additionally, repository technology could benefit other KBSE implementations such as Specware [42]. Specware is a software synthesis tool based on formal algebraic specifications. It uses category theory to create new specifications. Currently, many specifications are in a single directory structure in a surface syntax representation. Users must locate

appropriate specifications based solely on sequentially searching files for the one desired. Repository technology would significantly improve this process.

Use other repositories: Fully implemented, commercial repositories are on the market. It is unclear if these repositories can meet the specific needs of software synthesis. However, future researchers should implement the AWSOME model as an information model in one of these repositories. AFIT licenses several copies of Microsoft Visual Studio. The Microsoft repository is a component of the Visual Studio product and is based on the Microsoft Common Object Model (COM). The AWSOME information model could be developed in this repository and performance and reusability within a more generic, commercial repository could be evaluated with a fully-functional, software-synthesis-focused repository.

AWSOME extensions: Finally, the AWSOME model could be extended to capture additional semantics. Currently, only generic object-oriented design concepts can be represented in the AWSOME. To generate specific language constructs, it is often necessary to transform AWSOME abstract syntax trees into language-specific ASTs from which source code can be generated. It would be more efficient if more specific language constructs were incorporated into the AWSOME (a departure from the "core feature" focus of the current language). This could allow language-specific transformations to be retained in the AWSOME AST instead of transformed to a language-specific AST or parsed out to some surface syntax.

Appendix A. Building a Refine Interface to C

There is certainly more than one way to call external programs from a Refine program. However, we explicitly document the one used in this research.

A.1 Preparing the external program

1. Create C source files

   ```
   source1.c, source2.c, source3.c, ...
   ```

2. Code C functions, as normal

   ```
   int functName(int x)
   {
   return x + 5;
   }
   ```

   might be included in the source2.c file.

3. Compile C source files to object files

   ```
   gcc -c source1.c  source2.c source3.c
   ```

4. Link object files with -G option to produce a single shared object files. The .so extention is *required* .

   ```
   ld -G -o sharedObjectName.so source1.o source2.o source3.o
   ```

A.2 Define external functions into Lisp

1. Load shared object file
   `load (''sharedObjectName.so'')`
   Note: This is Refine syntax
2. Prepare Refine program to define functions to Lisp

   ```
   FF::defforeign( 'functName,
                   '::entry-point ''funtName'',
                   '::arguments, var-or- nil,
                   '::return-type, '::type-name);
   ```

3. Run Refine program that defines functions

A.3 Prepare Refine program to call external functions via Lisp

1. Add appropriate calls to external functions to Refine source

   ```
   function runit() =
   functName(5)
   ```

2. Compile Refine source in the normal manner
3. Run Refine program
   `(runit)`
   will return 10 in this example

A.4 Example

A.4.1 The file ctest.c includes:

```
#include <stdio.h>

main()
{
int i = 3;
printf("This is a test: %d\n", i);
}

receiveParm(j)
int j;
{
printf("Received parameter: %d \n", j);
}

int returnParm()
{
printf("Returning a 5\n");
return 5;
}

int exampleOperation(int x,int y)
{
printf("Returning %d\n", x + y);
return x + y;
}
```

A.4.2 The Refine file, calltest.re, defining functions includes: This is designed as a Refine function that calls the Lisp, `load` and `defforeign`. When this Refine program is executed, the respective C functions will be available for calling by Refine programs.

```
!! in-package("RU")
!! in-grammar('user)
```

```
function loadForeign()=
 Let (j:seq(symbol) = ['integer],
      k:seq(symbol) = ['integer, 'integer])
 load ("ctest.so");

  FF::defforeign('main,
                 '::arguments, nil,
                 '::return-type, '::void);
  FF::defforeign('receiveParm,
                 '::entry-point, "receiveParm",
                 '::arguments, j,
                 '::return-type, '::void);
  FF::defforeign('returnParm,
                 '::entry-point, "returnParm",
                 '::arguments, nil,
                 '::return-type, '::integer);
  FF::defforeign('exampleOperation,
                 '::entry-point, "exampleOperation",
                 '::arguments, k,
                 '::return-type, '::integer)
```

Note: No entry point is defined for the function, main()

A.4.3 Calling Refine program. This is an example Refine program that calls the external C funtions, passes parameters, and accepts results.

```
!! in-package("RU")
!! in-grammar('user)


function runForeign()=
  Let(a:integer = 3,
      b:integer = 4,
      c:integer = undefined,
      d:integer = undefined)
  main();
  format(true, "refine: called main...~%");
  receiveParm(8);
  format(true, "refine: called receiveParm...~%");
  format(true, "refine: called return parm.  Returned a: ~D ...~%",
                 returnParm());
  format(true, "refine: called return parm.  Expected to return a: ~D ...~%",
                 a + b);
```

```
  c <- exampleOperation(a,b);
  format(true, "refine: called exampleOperation. sent ~D and ~D and returned
        a ~D...~%", a,b, c)
```

A.4.4 Results of execution.

```
> (runForeign)
> This is a test: 3
> refine: called main...
> Received parameter: 8
> refine: called receiveParm
> refine: called return parm.  Returned a: Returning a 5
> 5
> refine: called return parm.  Expected to return a: 7
> Returning 7
> refine: called exampleOperation. Sent 3 and 4 and returned a 7
>
```

Appendix B. AWSOME

The AWSOME model, like its predecessors, was designed to capture the fundamental concepts of formal object-oriented modeling, as well as imperative and object-oriented programming languages. Several concepts of the AWSOME model have already been discussed in the context of demonstrating the methodology for combining the COIL and DOM. In particular, Chapter IV showed how the type system, packages, and classes were merged. This appendix describes the major concepts of the entire model.

B.1 The AWSOME inheritance diagram

The AWSOME inheritance hierarchy shows the derivation of every element of the AWSOME model. It can serve as a "table of contents" of the model. The inheritance diagram is shown in Figures 39, 40, 41.

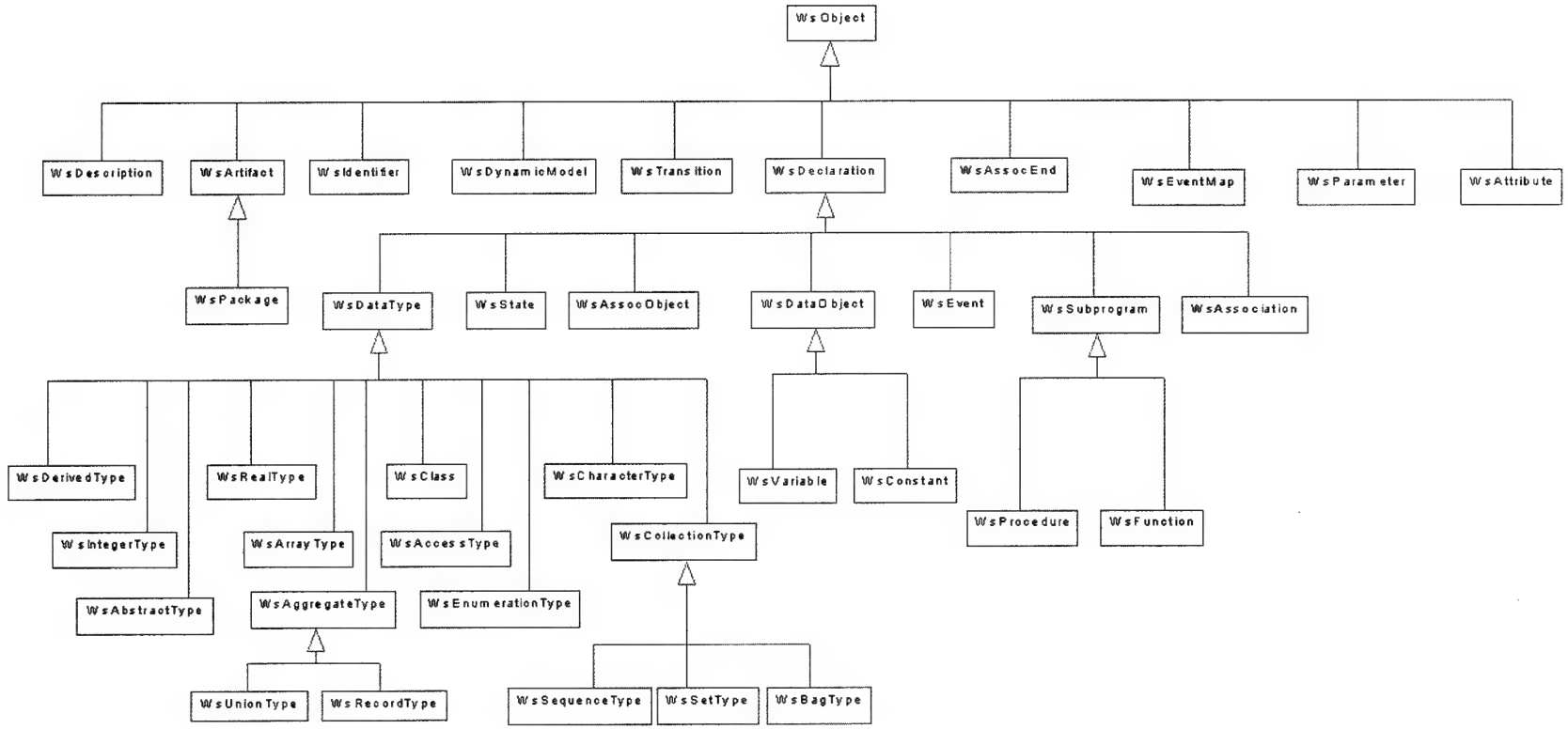

Figure 39. The AWSOME inheritance hierarchy (Part 1)

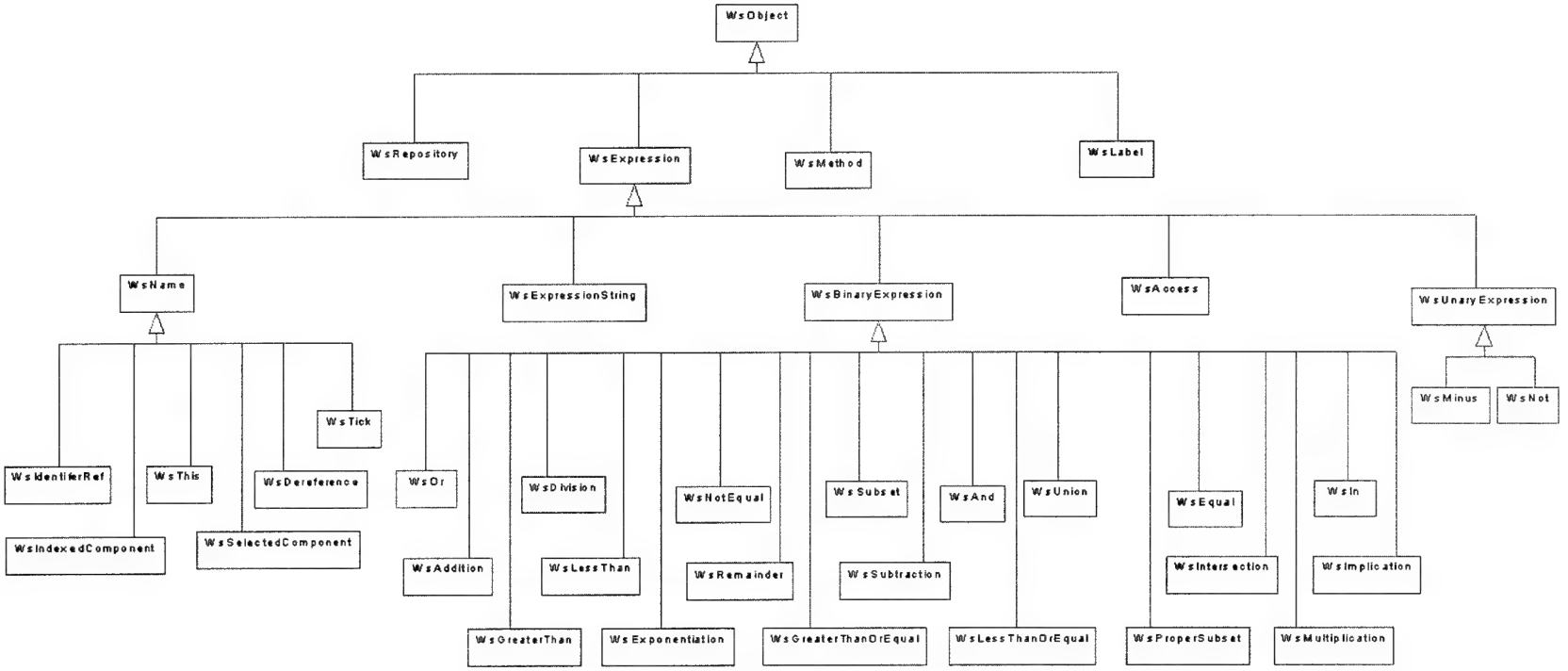

Figure 40. The AWSOME inheritance hierarchy (Part 2)

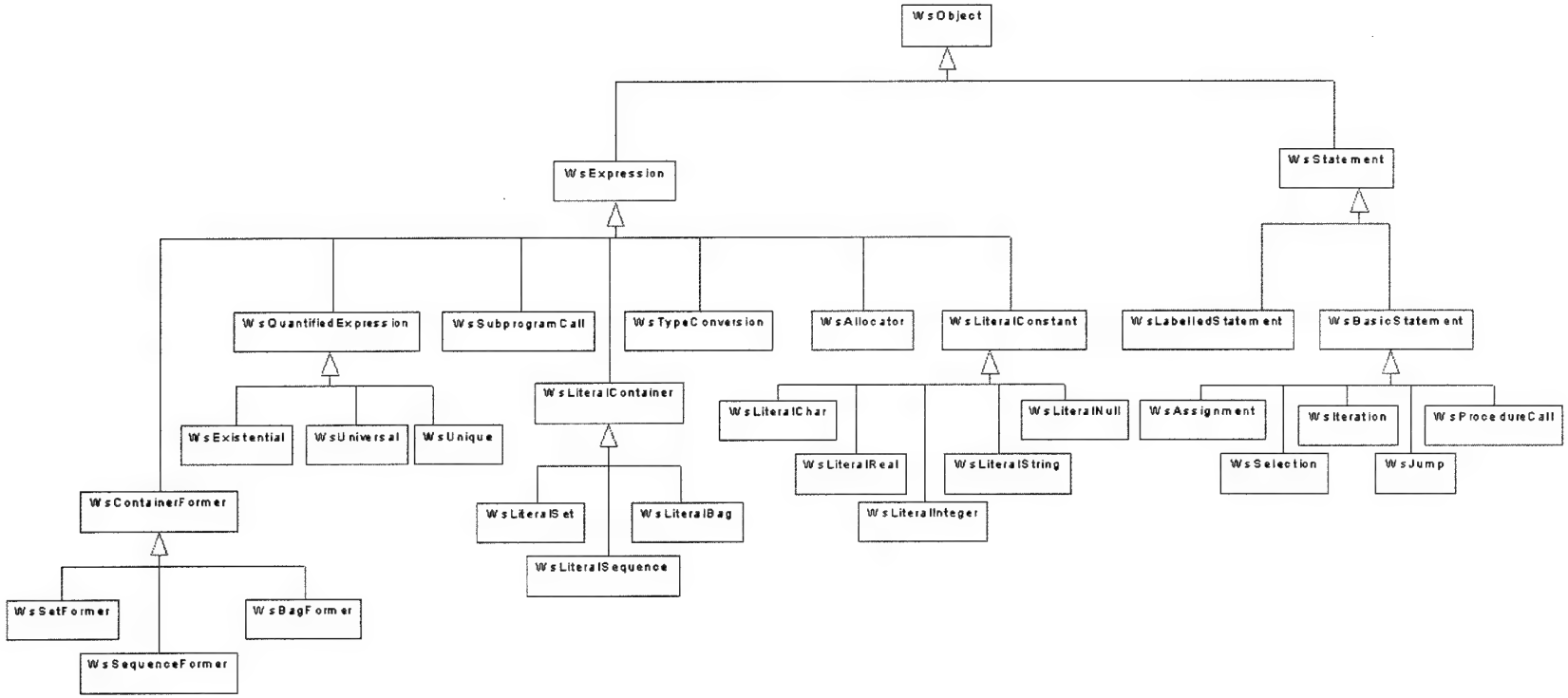

Figure 41. The AWSOME inheritance hierarchy (Part 3)

B.2 Key Components

B.2.1 Identifier. Undoubtedly *the* most important component of the AWSOME model is the identifier. WsIdentifier is a component of every named entity in the model. For example, declarations, parameters, and labels are all named entities. The identifier, WsIdentifier, represents the name of the declared item. The actual name of the item is represented as a component string of the WsIdentifier called wsIdentSymbol.

The remaining attribute of WsIdentifier mentioned here is wsDescription[1]. WsDescription will be addressed later in B.7. WsIdentifier is shown in Figure 42.

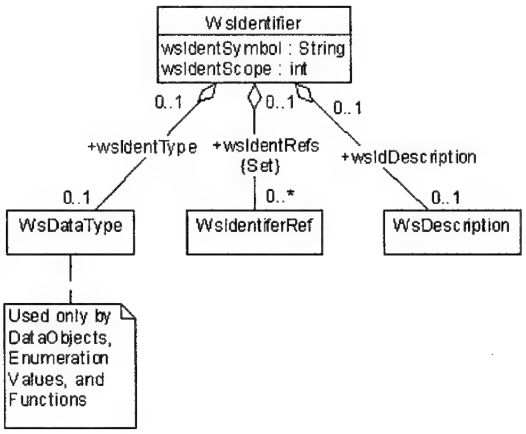

Figure 42. AWSOME Identifier

B.2.2 Object. The root of the AWSOME inheritance tree is WsObject. Every object in the AWSOME model inherits a parent attribute from WsObject. The parent attribute is a non-tree reference to the parent of each modeled entity. This attribute is populated when ASTs are built and is used extensively in tree navigation. WsObject also

[1]The attributes wsIdentRef, wsIdentScope, and wsIdentType are related to parsing, linking, and verification of ASTs and will be addressed in [48].

provides two Boolean attributes to every object in an AST. They can be used for marking visited nodes during tree navigation.

B.2.3 Visitor. One useful design pattern used in the construction of the AWSOME model is the visitor pattern [22]. The methodology provides a mechanism for programmers to implement new methods for AWSOME classes without actually modifying the class (since the visitor classes are implemented outside the AWSOME model). The visitor pattern accomplishes this by implementing a method, acceptVisitor, in every object in the model hierarchy. To perform operations on any given tree or subtree in the object hierarchy, a visitor is implemented. This visitor will contain the operations of the method that would have otherwise been added to the AWSOME model. The visitor method calls the acceptVisitor of a particular AWSOME class instance, the root of the tree or subtree. The built-in acceptVisitor method returns its own instance to the calling visitor. The visitor, then performs the desired operations on the class. Visitors have been used extensively in all tools developed for the AWSOME.

B.3 Declarations

The root of an AWSOME abstract syntax tree is a package. A package is defined as a set of declarations (as well as other packages). These declarations can be types, data objects, subprograms, associations, associative objects, and include dynamic model concepts such as states and events. Declarations can be fully defined inside the model, or can be "external" declarations. External declarations provide place-holders for types, data objects, etc . that might come from an external source, such as a third–party library.

External declarations are distinguished from those fully described in the model by setting a Boolean variable, wsExternal, to `true`. Finally, declarations have a name (a WsIdentifier). The component diagram for an AWSOME Declaration is shown in Figure 43. The kinds of AWSOME declarations are discussed in the next several sections.

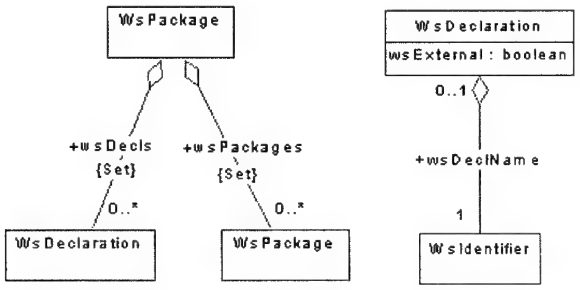

Figure 43. AWSOME Declarations

B.3.1 Types. The AWSOME type system was discussed in Section 4.1.1. However, this section will enumerate basic AWSOME types and discuss their components.

1. Derived

   A derived type is an aggregate of two objects. The first, called wsParent, is the type from which it was derived. The second, wsConstraint, are the constraints that further describe the new type (Figure 44). WsDerivedType inherits a name from WsDeclaration.

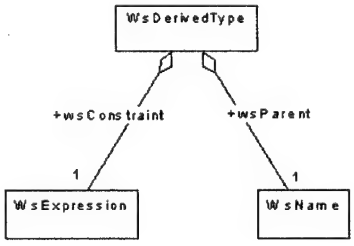

Figure 44. AWSOME Derived Types

2. Integer

   The integer type was described for the earlier example (Section 4.1.1.1) but is mentioned here for completeness.

3. Abstract

   Abstract types have no description and exist primarily to accommodate types defined outside the AWSOME model, that is, external types. Abstract types inherit an identifier from WsDelaration. This type enables an AWSOME AST to reference types not modeled in the AWSOME. An example might be classes defined in third-party libraries.

4. Real

   The real type is one of the more complex types of the AWSOME model. Like integer type, a real type has upper and lower bounds. Real types can be fixed-point or floating-point. Fixed-point numbers have a minimum step size. For money, this is 0.01. Floating-point numbers have a precision and, optionally, a base. The precision specifies the number of digits in the mantissa. Additionally, a floating-point type has a base—the default is 10, but 2, 8, or 16 are often used. Base defines the meaning of "digit." The component diagram for a real type is shown in Figure 45.

5. Access

   Access types represent traditional "pointers." Access type declarations contain a reference to the type of the object to which they point.

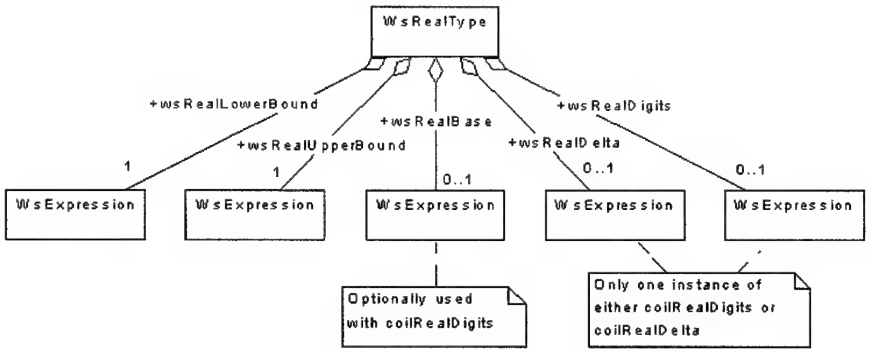

Figure 45. AWSOME Real Types

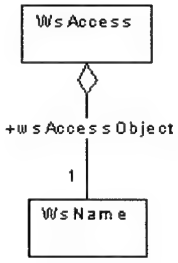

Figure 46. AWSOME Access Types

6. Aggregate

Record and union types are modeled in a structure called Aggregate Type. A record is a set of data objects and their types. The C `struct` is a record type. Unions allow a single data object to be referred to by different identifiers and types. The C `union` is an example of an AWSOME union type. An aggregate type is made up of a set of variables. The object diagram for the AWSOME Aggregate Type is shown in Figure 47.

7. Enumeration

Enumeration types allow type values to be explicitly listed. A simple example of an enumeration type is the AWSOME Boolean type. Following the paradigm of the

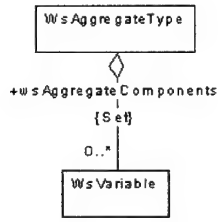

Figure 47. AWSOME Aggregate Types

COIL, Boolean is not a separate type, but an instance of an AWSOME Enumeration Type.

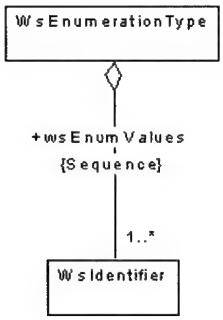

Figure 48. AWSOME Enumeration Types

8. Character

The AWSOME provides a character type. For simplicity of the implementation, the representation of a generic character has currently been defined as a Java `char` (see Item 4 of B.4).

9. Array

Array types are common in most modern third generation programming languages. In fact, implementation of collection types are often based on arrays. AWSOME arrays have a component index type. Though some programming languages restrict array indexes to integer types, some languages allow other index types. Some other

index types include enumeration or character types. The AWSOME model supports this by providing a component reference to the index type. The object diagram for an Array type is shown in Figure 49.

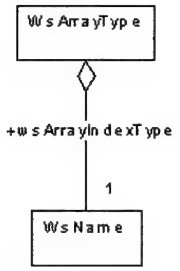

Figure 49. AWSOME Array Types

10. Collection

Collection types are becoming standard in many modern programming languages. Though truly "convenience features," collections provide enough utility to be included in a generic model. Additionally, many formal languages describe predicates in terms of the mathematical behavior of sets and sequences. Collection types are necessary to fully model these formally specified domain theories and requirements.

A collection is an aggregation of a WsIdentifier (inherited from WsDeclaration) and a reference to the type of the objects in the collection, a WsName. The object model for a collection type is shown in Figure 50.

(a) Sequence

A sequence type has no more structure than its parent collection type. Sequence type adds the additional semantics necessary for a sequence.

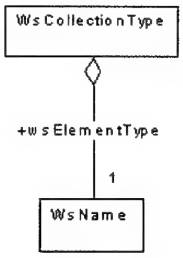

Figure 50. AWSOME Collection Types

(b) Set

A set type also retains the structure of a collection type. The set type will provide all the semantics for set theory.

(c) Bag

A bag type is also a collection type. As expected, it operates just like a set, with the exception of allowing duplicates.

B.3.2 Data Objects. The next AWSOME declaration discussed is the data object. A data object, like all declarations, has an identifier. The other components of a data object are the type, a reference (by name) to a WsDataType, and an expression. Subtypes of data items are variables and constants. In the case of a constant, the expression semantically represents the fixed value stored in the constant. For variables, the semantics of the expression is an initial value. Data Objects are shown in Figure 51.

B.3.2.1 Subprograms. The AWSOME model subprogram structure represents both procedures and functions. The original structure of subprogram is rooted in the COIL. However, additional semantics were added for the wide-spectrum nature of the AWSOME model (as discussed in Section 4.1.4).

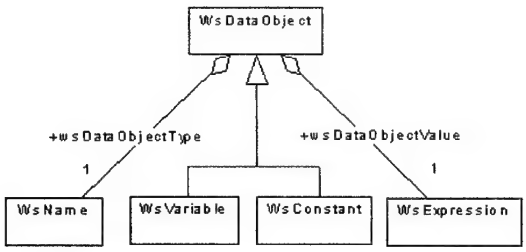

Figure 51. AWSOME Data Objects: Variables and Constants

The AST for a subprogram will differ, depending on the phase in the development lifecycle. A subprogram instance early in the process will be assigned the semantics of an operation in a formal specification. It will contain formal parameters, local constants, and pre- and post-conditions. The pre- and post-conditions will be expressed as predicates about the behavior of operations of a particular class. Pre- and post-conditions are implemented as expressions (expressions will be mentioned later in this appendix).

Later in the lifecycle, a subprogram is an instance of a design. It will consist of the same sequence of formal parameters, a set of locally declared variables and constants, and a sequence of program statements. Depending upon the implementation of a tool using this model, the design AST *may* also retain the specification data. This data may optionally be absent from the final design AST. The model of a subprogram is shown in Figure 52.

B.3.2.2 Parameter. An AWSOME model parameter, WsParameter, is used to define the formal parameters of a subprogram. They act like variable declarations, but have the added semantic of having a mode: `in`, `out`, or `in out`. The semantic of a mode is equivalent to that of Ada. The structure of a WsParameter is shown in Figure 53.

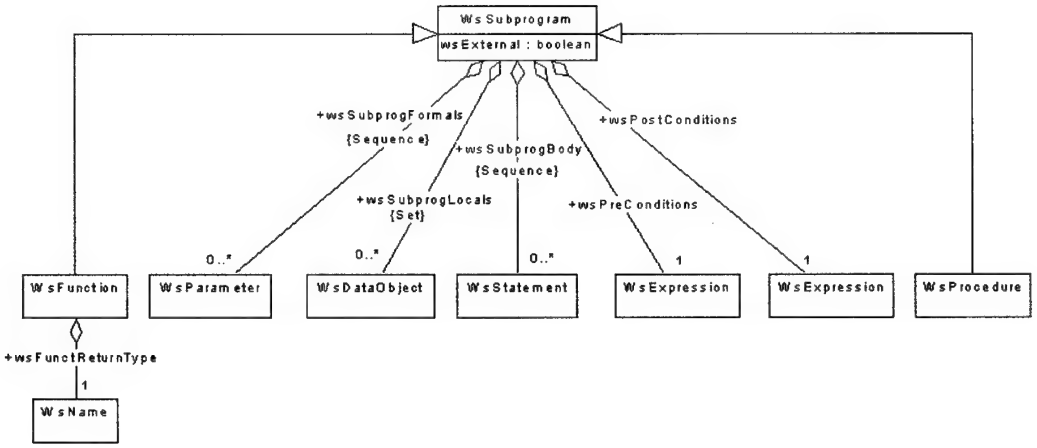

Figure 52. AWSOME Subprograms: Procedures and Functions

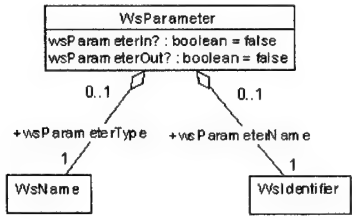

Figure 53. AWSOME Parameters

B.3.3 Classes. As mentioned earlier, a class behaves similarly to a type, so the AWSOME model treats it as a subtype of WsDataType. Part of the object diagram for a class is shown in Figure 29. A class has an inherited identifier. It also contains a set of attributes, a set of methods, and a reference to its superclass. To support object–oriented specification, the class also contains a dynamic model and a set of event maps.

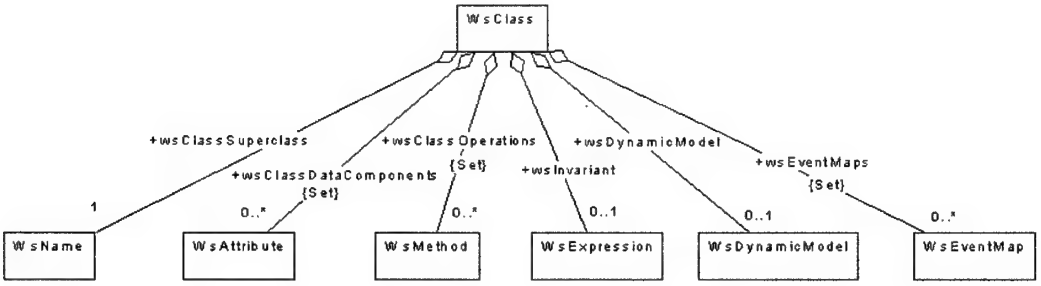

Figure 54. AWSOME Class

1. Superclass

   The superclass is referred to by the wsClassSuperclass attribute of a class. The wsClassSuperclass attribute is a reference, by name, to the parent class.

2. Attributes

   The attributes of a class are represented by a set of WsAttribute instances. Attributes contain Data Objects. Much of the structure of WsAttribute was created to support ongoing research in generating relational databases and SQL queries during software synthesis. Discussion of WsAttribute components related to the relational database research can be found in [11]. The structure of WsAttribute is shown in Figure 55.

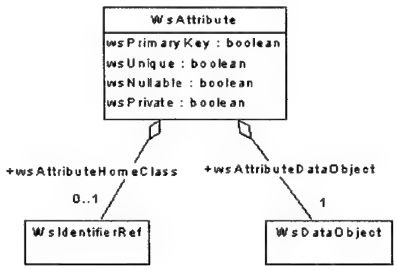

Figure 55. AWSOME Attribute

3. Methods

The methods of a class are represented as a set of WsMethod instances. The method is an abstraction of a subprogram (described above). WsMethod contains two attributes unique to object-oriented programming. As can be seen in Figure 56, there is a Boolean used to distinguish public from private methods. Another Boolean differentiates class-wide methods (a `static` in Java) from instance methods.

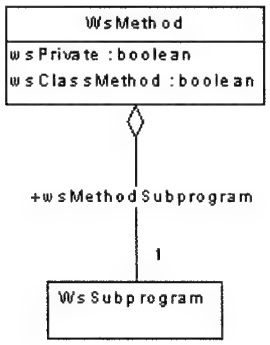

Figure 56. AWSOME Method

4. Invariant Constraints

Classes can have an invariant constraint. The AWSOME model captures this as a single expression of predicates joined by conjunction. For example, a formally expressed *Age* constraint might be stated like this:

BirthYear $\leq$ ThisYear $\wedge$
Age $\leq$ (ThisYear - BirthYear) $\wedge$
Age $\geq$ 0 $\wedge$
Age $\leq$ MAX_AGE;

This invariant is modeled by building a single expression. In the previous example syntax, the expression AST would have an instance of a kind of WsExpression, WsAnd, at the root, followed by a second level of the left hand binary expression,

WsLessThanOrEqualTo, and the right hand binary expression of another WsAnd, and so on. WsExpression will be discussed more in a later section.

5. Dynamic Model

The AWSOME Dynamic Model structure models the execution characteristics of the program or class. The dynamic model consists of sets of the AWSOME components WsEvent, WsTransition, and WsState. Details about the AWSOME Dynamic Model are in [33]. The structure of the Dynamic Model is shown in Figures 57, 58, 59 and 60.

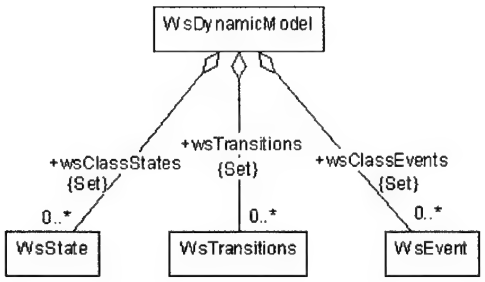

Figure 57. AWSOME Dynamic Model

(a) Event

An event is an occurrence that causes a transition between states.

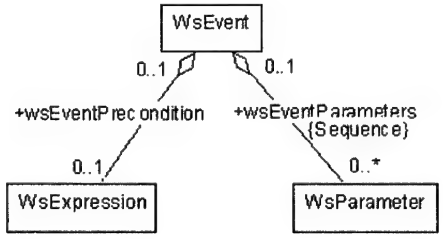

Figure 58. AWSOME Event

(b) Transitions

Transitions move a class from one state to another.

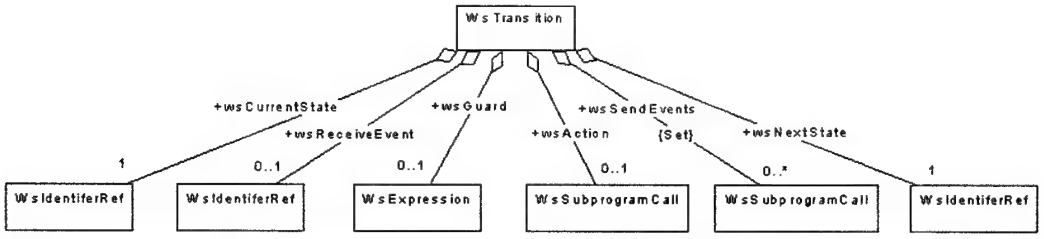

Figure 59. AWSOME Transitions

(c) State

WsState models declared states in which an object may exist.

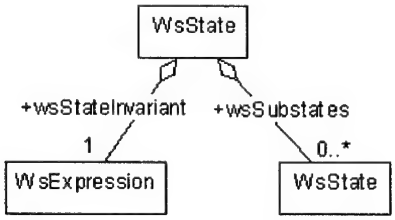

Figure 60. AWSOME State

6. WsEventMap

An event map is intended to represent the relationship between instances of objects that interact under the dynamic model. Ideally, the WsEventMap will be a component of an aggregate object and represent the dynamic interaction between its components. AFIT has not completed research on this aspect of the dynamic model, but the vision of what the event map might look like is shown in Figure 61.

B.3.3.1 Example AWSOME Class. Though the complete surface syntax for the AWSOME language is still under development, the surface syntax of the COIL can be

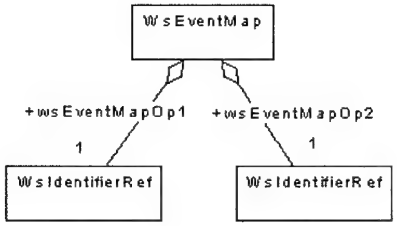

Figure 61. AWSOME Event Map

used along with a provisional syntax for the wide-spectrum extensions to specify a simple counter class. In this example, the type Integer is declared globally to reflect a typical 32-bit integer. A subtype of this integer, Natural, is also declared. Also, an enumeration type, CountMode, is declared. Finally, a constant, MAX_COUNT is also declared globally.

```
type Integer is range -2147483648 .. 2147483647;
subtype Natural is range 0 .. 2147483647 of Integer;
type CountMode is (up, down);
MAX_COUNT : const Integer := 99999;
class Counter is
    var count : Natural;
    var limit : Natural;
    var mode: ModeType;
    var margin: Natural;
    var maxReached : Natural;
    invariant
        count ≤ limit ∧
        maxReached ≤ limit ∧
        limit ≤ MAX_COUNT ∧
        margin = limit − count;

end class;
```

There are no operations in this class because it is a specification, not a design (the dynamic model has been omitted). Once this class was transformed by the software synthesis system, a number of operations for setting, resetting, incrementing, etcwould follow the invariants.

B.4 Expressions

Expressions were briefly discussed in Chapter IV. An extensive treatment of the vast expression hierarchy will be discussed in [48]. Thomson's research focuses on semantic verification of AWSOME ASTs. This requires an extensive discussion of AWSOME Expressions. The structure of expressions is shown in Figures 62 through 68.

1. Binary Expression

   A binary expression has a left and right side (also expressions). Example of a binary expressions are $+$, $-$, $\times$, $\wedge$, and $\vee$.

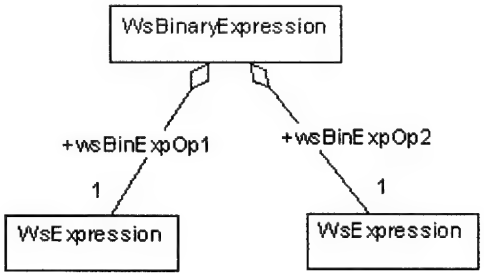

Figure 62. AWSOME Binary Expressions

2. Unary Expression

   A unary expression provides the capability to negate an expression with $-$ and $\neg$.

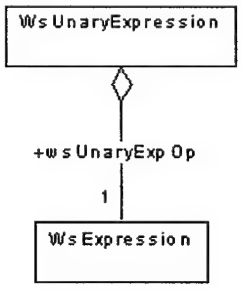

Figure 63. AWSOME Unary Expressions

3. Quantified Expression

A quantified expression is used extensively in the articulation of predicates. Quantified expression implement universal (∀), existential (∃), and unique (∃ !) quantifiers.

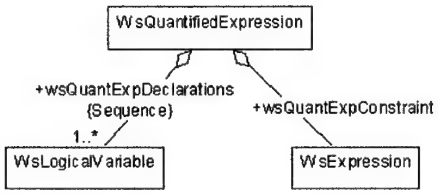

Figure 64. AWSOME Quantified Expressions

4. Literals

AWSOME literals store actual values in the model [24]. For example, the declaration of a real type will specify an upper and lower bound. The value of each of the bounds will be stored in separate instances of WsLiteralReal. Similarly a set, sequence, or bag will be stored in an instance of WsLiteralContainer [11].

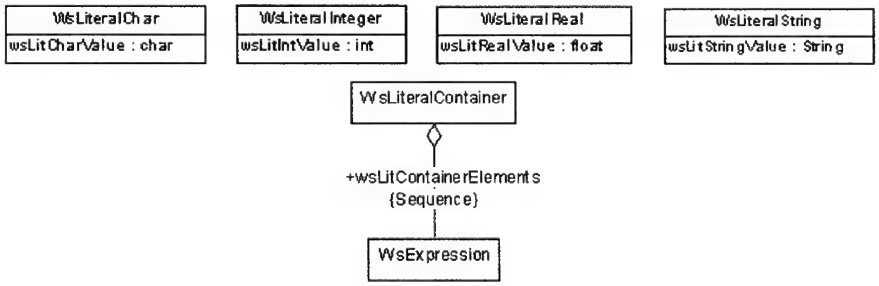

Figure 65. AWSOME Literals

5. Name References

AWSOME names are WsIdentifierRef, WsThis, WsTick, WsDereference, WsIndexedComponent, and WsSelectedComponent. A WsName is a reference to the WsIdentifier of a named entity. Some name references, such as records, can have their own

names while also having component named entities. To model this, a WsIdentifierRef instance will refer to the components of the record. As seen earlier, WsIdentifierRef was also used to point to the type declaration of a variable or attribute.

In addition to the identifier reference, WsDereference indicates a pointer dereference. A WsSelectedComponent represents the dot notation used in object-oriented and record component access (e.g. `recordName.recordField`). It can be used along with WsThis for self-referencing object instances (e.g. `this.methodName`). WsIndexedComponent refers to the elements of an array. One other WsName, WsTick, is a decoration used along with any WsName to indicate the mutability of variables in a formal specification (e.g. A'=succ(A)). More information on name references can be found in [24].

6. Set, Sequence, and Bag Former

Formers were designed and implemented by Buckwalter. More information on these valuable expressions can be found in his research [11].

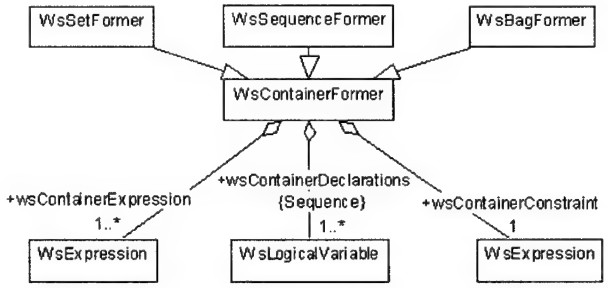

Figure 66. AWSOME "Formers"

7. WsSubprogramCall

A subprogram call is an expression that invokes a method or subprogram. It provides the calling parameters. It is also the sole component of a WsProcedureCall, a kind of statement. Procedure calls are discussed in B.6.

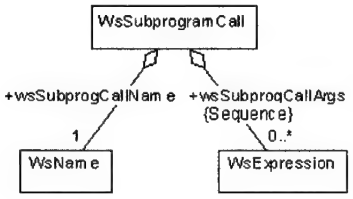

Figure 67. AWSOME Subprogram Call

8. Other Expressions

WsAccess and WsAllocator, like their COIL predecessors, allow the representation of pointer (WsAccessType) creation and dereference within the AWSOME model [24]. Their structure is shown in Figure 68.

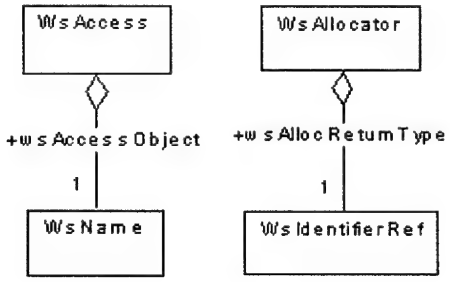

Figure 68. AWSOME Access and Allocator

B.5 Aggregation, Associations, and Associative Objects

In general, Associations are modeled with two or more association ends. Association Ends contain the multiplicity, role, and a reference to the class associated with the end.

Diagrammatically (in the Unified Modeling Language), the Association End (WsAssocEnd) represents the point where the line indicating an association meets the box indicating a class.

In the AWSOME model, aggregation is considered a special case of a binary association. Setting the Boolean attribute wsAggregate in the WsAssocEnd to `true` indicates that the class referred to by the WsAssocEnd is the aggregate, or "parent" class. The other WsAssocEnd in this binary association is the component class. WsAssociation and WsAssocEnd are in Figure 69.

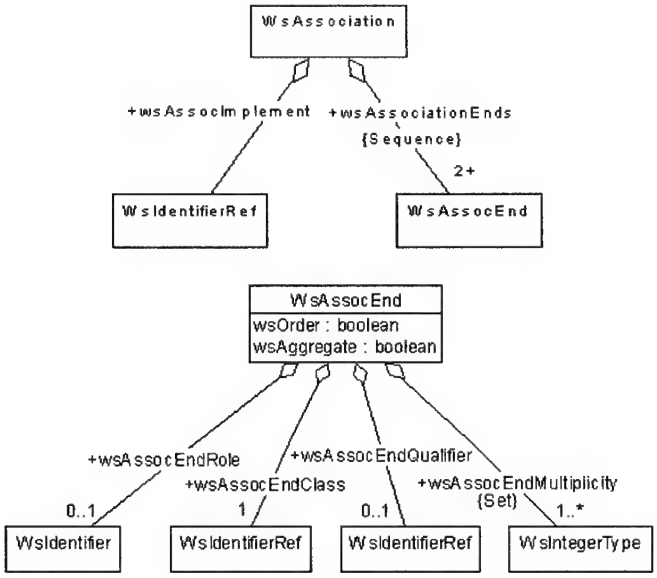

Figure 69. AWSOME Association and Association End

Associative Objects are modeled as a hybrid of an association and a class. Like classes they have attributes and operations. Like associations, they have two or more association ends. The structure of an associative object is given in Figure 70. The model of aggregation, association, and associative objects were designed by Buckwalter [11].

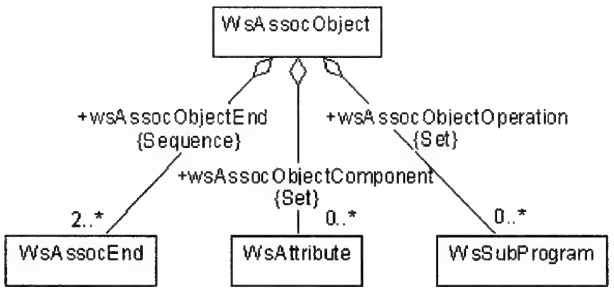

Figure 70. AWSOME Associative Object

B.6 Statements

AWSOME statements are no different than their COIL ancestors. They are shown in Figure 71. For a detailed explanation of the various kinds of statements, see [24].

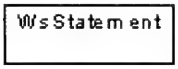

Figure 71. AWSOME Statements

1. Basic Statement

   Basic Statements include typical programming constructs: assignment, selection, iteration, procedure call, and jump[2]. The figures below show the structure of the different kinds of basic statements.

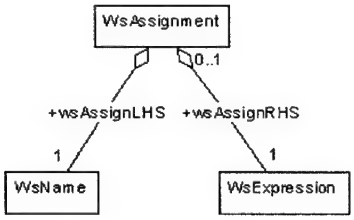

Figure 72. AWSOME Assignment Statement

[2]Yes, the jump is a `goto`. It was included in the language for the express purpose of achieving Dijkstra's goal of ridding the world of this evil [14]! It is only for parsing unstructured programs into the model so they can be reengineered to object-oriented (and have the `goto`s removed)!

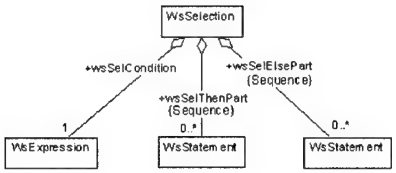

Figure 73. AWSOME Selection Statement

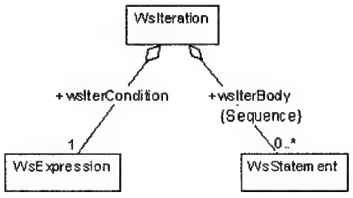

Figure 74. AWSOME Iteration Statements

AWSOME procedure calls are statements that contain the expression WsSubprogramCall. WsProcedureCall provides a "wrapper" for a subprogram call. This wrapper simply changes the context of the call from an expression to a statement. In the original COIL syntax, procedure call adds the terminating semicolon to the expression to transform it into a statement.

2. Labeled Statement

A labeled statement is nothing more than a basic statement with a label. Labeled statements are included to support the `goto` (the jump statement).

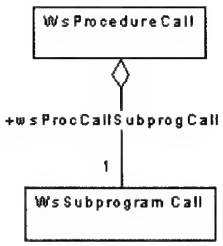

Figure 75. AWSOME Procedure Call Statement

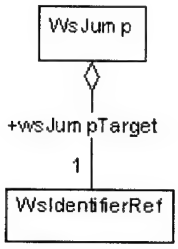

Figure 76. AWSOME Jump Statement

B.7 Odds and Ends

The AWSOME model contains several other classes that should be mentioned for completeness.

1. Label

   Labels are used in Fortran and other imperative languages. The AWSOME label supports the reengineering role the AWSOME model inherited from the COIL.

2. Repository

   The top-most aggregate object of the AWSOME model is WsRepository. This is included to make the model repository aware (see Section 5.4). It allows AWSOME to use more than one artifact (WsArtifact). A WsPackage is a WsArtifact and is sufficient to represent more than one AWSOME package; however, WsRepository and its collection of artifacts (WsArtifact) provide the capability for AWSOME tools to manipulate artifacts that may exist in other non-AWSOME models. For example, a tool might use an instance of WsRepository to contain an instance an AWSOME design AST and an instance of a text artifact to contain the source code generated from the design. The structure of WsRepository is shown in Figure 77.

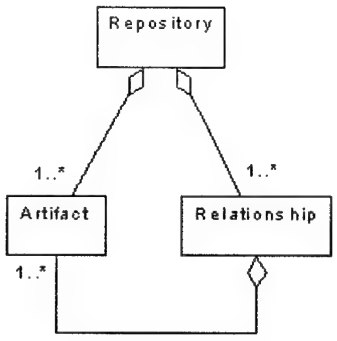

Figure 77. AWSOME Repository

3. Artifact

As just mentioned, the artifact (WsArtifact) is superclass of the package (WsPackage) in the AWSOME model. It provides a name and description to the artifact—each package instance. Artifact could also be implemented to represent other models, files, or binaries—as the WsRepository object intends[3]. The structure of WsArtifact is shown in Figure 78.

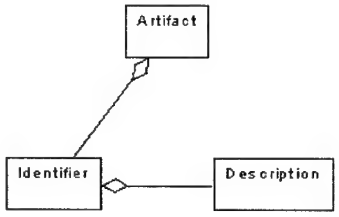

Figure 78. AWSOME Artifact

4. Description

Description is intended to provide meta-data about each identifier instance in the AWSOME model. Currently, the only attribute of a WsDescription is a string. A user

[3]To achieve the goal of representing arbitrary models or text items inside a tool, we contend WsArtifact should be implemented as a concrete class. However, other researchers have argued that it can be implemented as an abstract class for purposes of the current set of software synthesis tools.

is left to provide a free-text description. Eventually, some syntax for a description should be added to offer appropriate meta-data for information retrieval tools. These information retrieval tools will use the information in WsDescription and WsIdentifier to extract meta-data. This meta-data can aid the users in browsing, understanding, and selecting objects for reuse. The description object is shown in Figure 79.

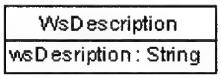

Figure 79. AWSOME Description

Bibliography

1. Anderson, Gary L. *An Interactive Tool for Refining Software Specifications from a Formal Domain Model*. MS thesis, Air Force Institute of Technology, 1999.

2. Ashby, Michael. *Tool-Based Integration and Code Generation of Object Models*. MS thesis, Air Force Institute of Technology, 2000.

3. Balzer, Robert. "Software Technology in the 1990's: Using a New Paradigm," *Computer* (1983).

4. Barber, K. Suzanne, et al. "Increasing Opportunities for Reuse Through Tool Methodology Support for Enterprise-wide Requirements Reuse and Evolution." *Proceedings of the 9th Workshop on Institutionalizing Software Reuse*. January 1999.

5. Bernstein, P.A., et al. "Microsoft Repository Version 2 and the Open Information Model," *Information Systems*, *24*(2) (1999).

6. Bernstein, Philip. "The Repository: A Modern Vision." *Database Programming and Design*. 28–35. December 1996.

7. Bernstein, Philip. "Repositories and Object Oriented Databases." *ACM SIGMOD Record*. March 1998.

8. Bernstein, Philip and T. Bergstrasser. "Meta-Data support for Data Transformation Using Microsoft Repository." *IEEE Data Engineering Bulliten22*. March 1999.

9. Bernstein, Philip and Umeshwar Dayal. "An Overview of Repository Technology." *Proceedings of the 20th International Conference on Very Large Database Systems*. 705–713. September 1994.

10. Biggerstaff, Ted. "The Library Scaling Problem and the Limits of Concrete Component Reuse." *Microsoft Reaseach MSR-TR-94-19*. 1994.

11. Buckwalter, Steven R. *Generating Executable Persistent Data Storage/Retrieval Code from Object-Oriented Specifications*. MS thesis, Air Force Institute of Technology, March 2000. AFIT/GCS/ENG/00M-02.

12. Colonese, Emilia. *Methodology For Integrating the Scenario Databases of Simulation Systems*. MS thesis, Air Force Institute of Technology, June 1999. AFIT/GCS/ENG/99J-03.

13. Constantopoulos, Panos, et al. "The Software Information Base: A Server for Reuse." *VLDB Journal*. 1–43. 1995.

14. Dahl, O.J., et al. *Structured Programming*. Academic Press, 1972.

15. deJesus Rodrigues, Sonia. *COBOL reengineering using the Parameter Based Object Identification (PBOI) methodology*. MS thesis, Air Force Institute of Technology, 1999.

16. DeLoach, Scott A. *Formal Transformations from Graphically-Based Object-Oriented Representations to Theory-Based Specification*. PhD dissertation, Air Force Institute of Technology, 1996.

17. Eichmann, David. "Supporting Multiple Domains in a Single Reuse Repository." *NASA-CR-190640*. 1992.

18. Etzkorn, Letha and Carl Davis. "Automatically Identifying Reusable OO Legacy Code," *Computer* (1997).

19. Frakes, W B and P B Gandel. "Representing Reusable Software," *Information and Software Technology* (1990).

20. Frakes, William and Thomas Pole. "An Empirical Study of Representation Methods for Reusable Software Components," *IEEE Transactions on Software Engineering* (1994).

21. Franz Inc. *Allegro CL Common Lisp, User Guide*, 1994.

22. Gamma, Erich, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

23. Graham, Robert P., "Class Lecture, COIL Presentation," 1999.

24. Graham, Robert P. "COIL Language Reference Manual (Draft)." Unpublished, Summer Quarter 1999.

25. Hartrum, T. and P. D. Bailor. "Teaching Formal Extensions of Informal-based Object-oriented Analysis Methodologies." *Software Engineering Education Proceedings (Pittsburg, PA), Software Engineering Education, SEI*. 1994.

26. Hartrum, Thomas. "An Object Oriented Formal Transformation System for Primitive Object Classes." Unpublished, 1999.

27. Hartrum, Thomas C., "Case Study: Simulation System Model," 1999.

28. Hartrum, Thomas C., "Class Notes, TrafficLight," 1999.

29. Henninger, Scott. "Supporting the Construction and Evolution of Component Repositories." *Proceedings of the International Conference on Software Engineering*. 1996.

30. Hunter, Richard. "Once Is Not Enough," *CIO Magazine* (1997).

31. Isakowitz, Tomas and Robert Kauffman. "Supporting Search for Reusable Software Objects," *IEEE Transactions on Software Engineering* (1996).

32. Kissack, John A. *Transforming Aggregate Object-Oriented Formal Specifications to Code*. MS thesis, Air Force Institute of Technology, 1999.

33. Marsh, David. *Formal Object State Model Transformations for Automated Agent System Synthesis*. MS thesis, Air Force Institute of Technology, 2000.

34. Mili, Rym, et al. "Storing and Retrieving Software Components, a Refinement-based System," *IEEE Transactions on Software Engineering* (1997).

35. Moraes, Dina Leite. *Transforming COBOL legacy software to a generic imperative model*. MS thesis, Air Force Institute of Technology, 1999.

36. Noe, Penelope Ann. *A Structured Approach to Software Tool Integration*. MS thesis, Air Force Institute of Technology, 1999.

37. Oren, Tuncer and Martin Hitz. "Requirements for a Repository-based Simulation Environment." *Proceedings of the 1992 Winter Simulation Conference*. March 1992.

38. Ostertag, Eduardo, et al. "Computing Similarity in a Reuse Library System: An AI-Based Approach." *ACM Transactions on Software Engineering Methodology*. 205–228. July 1992.

39. Parnas, David L. "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering* (1976).

40. Priato-Diaz, Reuben. "Domain Analysis for Reusability." *Proceedings of the IEEE COMPSAC '87*. 23–29. 1987.

41. Reasoning Systems Inc. *REFINE User's Guide*, 1995.

42. Reasoning Systems Inc. *SPECWARE User's Guide*, 1995.

43. Sen, Arun. "The Role of Opportunism in the Software Design Reuse Process," *IEEE Transactions on Software Engineering*, 418–436 (July 1997).

44. Simulation Technology Branch. *CERTCORT Requirements Document*, August 1997. Systems Concepts and Simulation Division, Avionics Directorate, Wright Laboratory, Air Force Materiel Command, Wright-Patterson Air Force Base, OH.

45. Smith, D. R. "KIDS - A Semi-automatic Program Development System," *IEEE Transactions on Software Engineering* (1990).

46. Sward, Rickey E. *Extracting Functionally Equivalent Object-Oriented Designs from Imperative Legacy Code*. PhD dissertation, Air Force Institute of Technology, 1997.

47. Tankersley, Travis W. *Generating Executable Code from Formal Specifications*. MS thesis, Air Force Institute of Technology, 1999.

48. Thomson, Steven. *Validation and Verification of Formal Specifications in Object-Oriented Software Engineering*. MS thesis, Air Force Institute of Technology, 2000.

49. Vijayaraman, T. M., et al., editors. *Modeling Design Versions*, Morgan Kaufmann, 1996.

50. Williams, Darin. *ExplicitlyModeling Hierarchically Heterogeneous Software Architectures in an Object-Oriented Formal Transformation System*. MS thesis, Air Force Institute of Technology, 2000.

51. Wirth, N. "Program Development By Stepwise Refinement," *Communications of the ACM* (1971).

Vita

Captain Cornn was born on 9 March 1964 in Boynton Beach, Florida. He received his Bachelor of Science in computer science from Florida Atlantic University in Boca Raton, Florida. He was commissioned as a Second Lieutenant in 1991 after completing Air Force Officer Training School. After completing Basic Communications Officer Training, he served as a database programmer/analyst in the War Planning Systems Division, and later as a technical manager in the systems engineering branch of the Strategic War Planning Systems Program Office, United States Strategic Command, Offutt AFB, Nebraska. While there, he did graduate work at University of Nebraska at Omaha. Next, he led on-site communications systems integration efforts for the Air Force Satellite Control Network Program Office and later led the Communications, Range, and Support Directorate of Detachment 2, Space Systems Support Group at Schriever AFB, Colorado. He has earned Joint Commendation and Air Force Commendation Medals.

He is married to the former Cheri Boiseau, of West Palm Beach, Florida. They have two beautiful children: Tiffany and Allison.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 2001	Master's Thesis

4. TITLE AND SUBTITLE

An Object-Oriented Repository-based Software Synthesis System

5. FUNDING NUMBERS

6. AUTHOR(S)

Gary L. Comn, Jr, Captain, USAF

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 P Street, Building 640
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/00M-05

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/IFTD
Attn: Roy F. Stratton
525 Brooks Rd
Rome, NY 13441-4505
(330) 315-3004 DSN: 587-3004

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

Maj Robert P. Graham, Jr., ENG, DSN: 785-3636, ext. 4595 Robert.Graham@afit.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

ABSTRACT (*Maximum 200 Words*)

This research provides a repository on which various Air Force Institute of Technology (AFIT) transformational software synthesis tools can store, share, and manage data using a common repository information model. This information model was created by integrating a variety of separately-developed AFIT software synthesis object models into a ``wide-spectrum" model. Additionally, a methodology for describing complex relationships between artifacts in the repository is described. These relationships can be used to relate software synthesis artifacts created in a variety of formats, including text, binary, and the AFIT Wide-Spectrum Object Modeling Environment (AWSOME) information model. The relationships can be exploited for the retrieval, understanding, and selection of reusable software engineering artifacts. Finally, a methodology that uses the repository relationships to generate a history of the semi-automatically generated designs is described. Future efforts can use the design history to re-create designs automatically when new requirements dictate changes to a related analysis model.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Repository, Software Synthesis, Transformation System, Design History, Object-Oriented, OODBMS, Software Engineering, Domain Theory, Reuse, Reusability	152
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements.***

Block 1. Agency Use Only *(Leave blank).*

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C - Contract	**PR** - Project
G - Grant	**TA** - Task
PE - Program Element	**WU** - Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with....; Trans. of....; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- **DOD** - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

Block 12b. Distribution Code.

- **DOD** - Leave blank.
- **DOE** - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- **NASA** - Leave blank.
- **NTIS** - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only).*

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.